Lecture 21: Congestion Control

CSE 123: Computer Networks
Aaron Schulman
CC Overview

- How fast should a sending host transmit data?
  - Not too fast, not too slow, just right...

- Should not be faster than the sender’s share
  - Bandwidth allocation

- Should not be faster than the network can process
  - Congestion control

- Congestion control & bandwidth allocation are separate ideas, but frequently combined
How much bandwidth should each flow from a source to a destination receive when they compete for resources?

- What is a “fair” allocation?
Buffer intended to absorb bursts when input rate > output
But if sending rate is persistently > drain rate, queue builds
Dropped packets represent wasted work; goodput < throughput
Drop-Tail Queuing

Loss due to Congestion

Congestion collapse

Throughput vs. Network Load

Latency vs. Network Load

Loss due to Congestion
Congestion Collapse

- Rough definition: “When an increase in network load produces a decrease in useful work”

- Why does it happen?
  - Sender sends faster than bottleneck link speed
  - Packets queue until dropped
  - In response to packets being dropped, sender retransmits
  - All hosts repeat in steady state…
Mitigation Options

- Increase network resources
  - More buffers for queuing
  - Increase link speed
  - Pros/Cons of these approaches?

- Reduce network load
  - Send data more slowly
  - How much more slowly?
  - When to slow down?
Designing a Control

- **Open loop**
  - Explicitly reserve bandwidth in the network in advance

- **Closed loop**
  - Respond to feedback and adjust bandwidth allocation

- **Network-based**
  - Network implements and enforces bandwidth allocation

- **Host-based**
  - Hosts are responsible for controlling their sending rate
**Proactive vs. Reactive**

- **Congestion avoidance**: try to stay to the left of the knee
- **Congestion control**: try to stay to the left of the cliff
Challenges to Address

- How to detect congestion?
- How to limit sending data rate?
- How fast to send?
Explicit congestion signaling
- Source Quench: ICMP message from router to sender
- DECBit / Explicit Congestion Notification (ECN):
  - Router marks packet based on queue occupancy (i.e. indication that packet encountered congestion along the way)
  - Receiver tells sender if queues are getting too full

Implicit congestion signaling
- Packet loss
  - Assume congestion is primary source of packet loss
  - Lost packets indicate congestion
- Packet delay
  - Round-trip time increases as packets queue
  - Packet inter-arrival time is a function of bottleneck link
Throttling Options

- **Window-based (TCP)**
  - Constrain number of outstanding packets allowed in network
  - Increase window to send faster; decrease to send slower
  - Pro: Cheap to implement (already have windowing), good failure properties
  - Con: Creates traffic bursts (requires bigger buffers)

- **Rate-based (many streaming media protocols)**
  - Two parameters (period, packets)
  - Allow sending of x packets in period y
  - Pro: smooth traffic
  - Con: fine-grained per-connection timers, what if receiver fails?
Choosing a Send Rate

- Ideally: Keep equilibrium at “knee” of power curve
  - Find “knee” somehow
  - Keep number of packets “in flight” the same
  - Don’t send a new packet into the network until you know one has left (i.e. by receiving an ACK)
  - What if you guess wrong, or if bandwidth availability changes?

- Compromise: adaptive approximation
  - If congestion signaled, reduce sending rate by $x$
  - If data delivered successfully, increase sending rate by $y$
  - How to relate $x$ and $y$? Most choices don’t converge…
TCP’s Probing Approach

- Each source independently probes the network to determine how much bandwidth is available
  - Changes over time, since everyone does this
- Assume that packet loss implies congestion
  - Since errors are rare; also, requires no support from routers
Window-based congestion control

- Unified congestion control and flow control mechanism
- \( rwin \): advertised flow control window from receiver
- \( cwnd \): congestion control window
  - Estimate of how much outstanding data network can deliver in a round-trip time
- Sender can only send \( \text{MIN}(rwin,cwnd) \) at any time

Idea: decrease \( cwnd \) when congestion is encountered; increase \( cwnd \) otherwise

- Question: how much to adjust?
Goal: Adapt to changes in available bandwidth

Additive Increase, Multiplicative Decrease (AIMD)
- Increase sending rate by a constant (e.g. MSS)
- Decrease sending rate by a linear factor (e.g. divide by 2)

Rough intuition for why this works
- Let $L_i$ be queue length at time $i$
- In steady state: $L_i = N$, where $N$ is a constant
- During congestion, $L_i = N + yL_{i-1}$, where $y > 0$
- Consequence: queue size increases multiplicatively
  » Must reduce sending rate multiplicatively as well
TCP Bandwidth Probing

- TCP uses AIMD to adjust congestion window
  - Converges to fair share of bottleneck link
  - Increases modestly in good times
  - Cuts drastically in bad times

- But what rate should a TCP flow use initially?
  - Need some initial congestion window
  - We’d like to TCP to work on all manner of links
  - Need to span 6+ orders of magnitude, e.g., 10 K to 10 Gbps.
  - Starting too fast is catastrophic!
Goal: quickly find the equilibrium sending rate

Quickly increase sending rate until congestion detected
  - Remember last rate that worked and don’t overshoot it

TCP Reno Algorithm:
  - On new connection, or after timeout, set $cwnd=1$ MSS
  - For each segment acknowledged, increment $cwnd$ by 1 MSS
  - If timeout then divide $cwnd$ by 2, and set $ssthresh = cwnd$
  - If $cwnd >= ssthresh$ then exit slow start

Why called slow? Its exponential after all…
Slow Start Example

CSE 123 – Lecture 20: Scheduling & QoS
Basic Mechanisms

Slow Start + Congestion Avoidance

- **cwnd**: congestion window
- **timeout**: time before reducing the congestion window
- **ssthresh**: slow start threshold
- **slow start**: initial phase of congestion control
- **congestion avoidance**: phase after slow start
Fast Retransmit & Recovery

- **Fast retransmit**
  - Timeouts are slow (default often 200 ms or 1 second)
  - When packet is lost, receiver still ACKs last in-order packet
  - Use 3 duplicate ACKs to indicate a loss; detect losses quickly
    » Why 3? When wouldn’t this work?

- **Fast recovery**
  - Goal: avoid stalling after loss
  - If there are still ACKs coming in, then no need for slow start
  - If a packet has made it through -> we can send another one
  - Divide $cwnd$ by 2 after fast retransmit
  - Increment $cwnd$ by 1 MSS for each additional duplicate ACK
Fast Retransmit Example

Fast recovery (increase cwnd by 1)

Fast retransmit
More Sophistication

Slow Start + Congestion Avoidance + Fast Retransmit + Fast Recovery

CSE 123 – Lecture 21: Congestion control
Short Connections

- Short connection: only contains a few pkts
- How do short connections and Slow-Start interact?
  - What happens when a packet is lost during Slow-Start?
  - What happens when the SYN is dropped?
- Bottom line: Which packet gets dropped matters a lot
  - SYN
  - Slow-Start
  - Congestion avoidance
- Do you think most flows are short or long?
- Do you think most traffic is in short flows or long flows?
TCP is designed around the premise of cooperation

- What happens to TCP if it competes with a UDP flow?
- What if we divide $cwnd$ by 3 instead of 2 after a loss?

There are a bunch of magic numbers

- Decrease by 2x, increase by $1/cwnd$, 3 duplicate acks, initial timeout = 3 seconds, etc.

But overall it works really well!

- Still being constantly tweaked…
TCP CC Summary

- TCP Probes the network for bandwidth, assuming that loss signals congestion.

- The congestion window is managed with an additive increase/multiplicative decrease policy:
  - It took fast retransmit and fast recovery to get there.
  - Fast recovery keeps pipe “full” while recovering from a loss.

- Slow start is used to avoid lengthy initial delays:
  - Ramp up to near target rate, then switch to AIMD.
For next time...

- RP&D 6.5
- Keep going on Project 2