Computer Network Fundamentals
Spring 2008

Week 10
Congestion Control
Andreas Terzis
Outline

- Congestion Control
  - TCP Congestion Control
What We Know

We know:

• How to process packets in a switch
• How to route packets in the network
• How to send packets reliably

We don’t know:

• How fast to send
What’s at Stake?

- Send too slow: link is not fully utilized
  - wastes time
- Send too fast: link is fully utilized but....
  - queue builds up in router buffer (delay)
  - overflow buffers in routers
  - overflow buffers in receiving host (ignore)
- Why are buffer overflows a problem?
  - packet drops (mine and others)
Abstract View

- We ignore internal structure of router and model it as having a single queue for a particular input-output pair
Three Congestion Control Problems

- Adjusting to bottleneck bandwidth
- Adjusting to variations in bandwidth
- Sharing bandwidth between flows
Single Flow, Fixed Bandwidth

- Adjust rate to match bottleneck bandwidth
  - without any *a priori* knowledge
  - could be gigabit link, could be a modem

![Diagram showing a single flow with fixed bandwidth](image-url)
Single Flow, Varying Bandwidth

- Adjust rate to match instantaneous bandwidth
  - assuming you have rough idea of bandwidth
Multiple Flows

Two Issues:
• Adjust total sending rate to match bandwidth
• Allocation of bandwidth between flows
Congestion control is a resource allocation problem involving many flows, many links, and complicated global dynamics.
General Approaches

• Send without care
  – many packet drops
  – not as stupid as it seems

• Reservations
  – pre-arrange bandwidth allocations
  – requires negotiation before sending packets
  – low utilization

• Pricing
  – don’t drop packets for the high-bidders
  – requires payment model
General Approaches (cont’d)

• Dynamic Adjustment
  – probe network to test level of congestion
  – speed up when no congestion
  – slow down when congestion
  – suboptimal, messy dynamics, simple to implement

• All three techniques have their place
  – but for generic Internet usage, dynamic adjustment is the most appropriate
  – due to pricing structure, traffic characteristics, and good citizenship
TCP Congestion Control

- TCP connection has window
  - controls number of unacknowledged packets

- Sending rate: ~Window/RTT

- Vary window size to control sending rate
Congestion Window \((cwnd)\)

- Limits how much data can be in transit
- Implemented as # of bytes
- Described as # packets in this lecture

\[
\text{MaxWindow} = \min(cwnd, \text{AdvertisedWindow})
\]

\[
\text{EffectiveWindow} = \text{MaxWindow} - (\text{LastByteSent} - \text{LastByteAcked})
\]
Two Basic Components

- Detecting congestion

- Rate adjustment algorithm
  - depends on congestion or not
  - three subproblems within adjustment problem
    - finding fixed bandwidth
    - adjusting to bandwidth variations
    - sharing bandwidth
Detecting Congestion

- Packet dropping is best sign of congestion
  - delay-based methods are hard and risky

- How do you detect packet drops? ACKs
  - TCP uses ACKs to signal receipt of data
  - ACK denotes last contiguous byte received
    - actually, ACKs indicate next segment expected

- Two signs of packet drops
  - No ACK after certain time interval: time-out
  - Several duplicate ACKs (ignore for now)
Rate Adjustment

• Basic structure:
  – Upon receipt of ACK (of new data): increase rate
  – Upon detection of loss: decrease rate

• But what increase/decrease functions should we use?
  – Depends on what problem we are solving
Problem #1: Single Flow, Fixed BW

- Want to get a first-order estimate of the available bandwidth
  - Assume bandwidth is fixed
  - Ignore presence of other flows
- Want to start slow, but rapidly increase rate until packet drop occurs (“slow-start”)
- Adjustment:
  - cwnd initially set to 1
  - cwnd++ upon receipt of ACK
Slow-Start

- cwnd increases exponentially: cwnd doubles every time a full cwnd of packets has been sent
  - Each ACK releases two packets
  - Slow-start is called “slow” because of starting point
Problems with Slow-Start

• Slow-start can result in many losses
  – roughly the size of cwnd $\sim$ BW*RTT

• Example:
  – at some point, cwnd is enough to fill “pipe”
  – after another RTT, cwnd is double its previous value
  – all the excess packets are dropped!

• Therefore, need a more gentle adjustment algorithm once have rough estimate of bandwidth
Problem #2: Single Flow, Varying BW

- Want to be able to track available bandwidth, oscillating around its current value
- Possible variations: (in terms of RTTs)
  - multiplicative increase or decrease: \( cwnd \rightarrow a \times cwnd \)
  - additive increase or decrease: \( cwnd \rightarrow cwnd + b \)
- Four alternatives:
  - AIAD: gentle increase, gentle decrease
  - AIMD: gentle increase, drastic decrease
  - MIAD: drastic increase, gentle decrease (too many losses)
  - MIMD: drastic increase and decrease
Problem #3: Multiple Flows

• Want steady state to be “fair”
• Many notions of fairness, but here all we require is that two identical flows end up with the same bandwidth
  – This eliminates MIMD and AIAD
    • Look at later explanation
  – AIMD is the only remaining solution!
Buffer and Window Dynamics

- No congestion $\rightarrow$ $x$ increases by one packet/RTT every RTT
- Congestion $\rightarrow$ decrease $x$ by factor 2
AIMD Sharing Dynamics

- No congestion $\rightarrow$ rate increases by one packet/RTT every RTT
- Congestion $\rightarrow$ decrease rate by factor 2

Rates equalize $\rightarrow$ fair share
AIAD Sharing Dynamics

- No congestion $\rightarrow$ $x$ increases by one packet/RTT every RTT
- Congestion $\rightarrow$ decrease $x$ by 1
AIMD

Limit rates: $x = y$
AIAD

Limit rates: 
x and y depend on initial values
Implementing AIMD

• After each ACK
  – increment $cwnd$ by $1/cwnd$ ($cwnd += 1/cwnd$)
  – as a result, $cwnd$ is increased by one only if all segments in a $cwnd$ have been acknowledged

• But need to decide when to leave slow-start and enter AIMD
  ▪ use ssthresh variable
Slow Start/AIMD Pseudocode

Initially:
   cwnd = 1;
   ssthresh = infinite;

New ack received:
   if (cwnd < ssthresh)
      /* Slow Start*/
      cwnd = cwnd + 1;
   else
      /* Congestion Avoidance */
      cwnd = cwnd + 1/cwnd;

Timeout:
   /* Multiplicative decrease */
   ssthresh = cwnd/2;
   cwnd = 1;
The big picture (with timeouts)

- **Timeout**
- ** AIMD**
- ** ssthresh**
- ** Slow Start**
- ** cwnd**

---

CS 344/Spring08
Congestion Detection Revisited

- Wait for Retransmission Time Out (RTO)
  - RTO kills throughput
- In BSD TCP implementations, RTO is usually more than 500ms
  - the granularity of RTT estimate is 500 ms
  - retransmission timeout is RTT + 4 * mean_deviation
- Solution: Don’t wait for RTO to expire
Fast Retransmits

- Resend a segment after 3 duplicate ACKs
  - A duplicate ACK means that an out-of-sequence segment was received

- Notes:
  - ACKs are for next expected packet
  - Packet reordering can cause duplicate ACKs
  - Window may be too small to get enough duplicate ACKs
Fast Recovery:
After a Fast Retransmit

- \( ssthresh = \frac{cwnd}{2} \)
- \( cwnd = ssthresh \)
  - instead of setting \( cwnd \) to 1, cut \( cwnd \) in half (multiplicative decrease)
- for each dup ack arrival
  - \( dupack++ \)
  - MaxWindow = min\( (cwnd + dupack, AdvWin) \)
  - indicates packet left network, so we may be able to send more
- receive ack for new data (beyond initial dup ack)
  - \( dupack = 0 \)
  - exit fast recovery
- But when RTO expires still do \( cwnd = 1 \)
Fast Retransmit and Fast Recovery

- Retransmit after 3 duplicated acks
  - Prevent expensive timeouts
- Reduce slow starts
- At steady state, $cwnd$ oscillates around the optimal window size
TCP Congestion Control Summary

- Measure available bandwidth
  - slow start: fast, hard on network
  - AIMD: slow, gentle on network

- Detecting congestion
  - timeout based on RTT
    - robust, causes low throughput
  - Fast Retransmit: avoids timeouts when few packets lost
    - can be fooled, maintains high throughput

- Recovering from loss
  - Fast recovery: don’t set cwnd=1 with fast retransmits
Issues to Think About

- What about short flows? (setting initial cwnd)
  - most flows are short
  - most bytes are in long flows

- How does this work over wireless links?
  - packet reordering fools fast retransmit
  - loss not always congestion related

- High speeds?
  - to reach 10gbps, packet losses occur every 90 minutes!

- Why are losses bad?
  - Tornado codes: can reconstruct data proportional to packets that get through. Why not send at maximal rate?

- Fairness: how do flows with different RTTs share link?