Announcements:
- CS grad recruitment event Wednesday November 16th, 2016 5pm - 7pm Rm 130 CSB
- Project 3 is still due on November 29th

TCP

Many modifications suggested to make TCP better
- Receiver should get good throughput
- Delay from sender side should not be large

TCP Nagle’s algorithm
- TCP handling small size packets
- Motivation:
  - Interactive applications (telnet, SSH, rlogin) generate many small packets (like keystrokes) that need to be sent to the receiver. Each packet needs a header
  - But, small packets are wasteful because most of the packet is the header
    - This wastes a lot of bandwidth
  - We want to reduce the number of packets sent.
    - We can make a minimum size for each of the packets: this can be wasteful
  - We want to balance the trade-offs
    - Send larger packets to increase efficiency and not increase delay
- Nagle’s Algorithm
  - If packet is smaller than min size and a packet is already in flight, we wait to sent the packet
  - So we send at most one small packet per RTT by waiting for all outstanding packets

Delayed ACK
- Motivation:
  - TCP traffic is often bidirectional
  - ACK packets have high overhead,
  - Send other data along with the ACK: Piggyback on the ACK packet to not be as wasteful
    - Example in the slides
      - A sends Data to B
      - B has data to send so it sends the ACK+Data
      - If B doesn’t have data to send, it will send the ACK without data
  - Increase the likelihood of Piggybacking:
- If B doesn't have data to send, a timer will start to wait for data. If B gets data to send, it will send the ACK + Data. If the timer runs out and B doesn't have data, it will just send the ACK

- Algorithm:
  - When B receives a packet, it sets a timer
  - If B generates data, send ACK + Data
  - If timer expires, send ACK without data (no piggy backing)

Faster TCP? TCP Throughput
- The average throughput of TCP as a function of window size and RTT
  - Assume a long-lived TCP flow
  - Ignore slow start
  - Start with window size W
  - Average throughput on receiver side is W/RTT
  - Just after loss of packet window drops to W/2, throughput to W/2RTT
    - Any loss we have is from congestion
  - Average throughput: 0.75 W/RTT

Problems with Fast Links
- 1500 byte segments
- 100ms RTT
- 10 Gb/s throughput
- Required window size?
  - Throughput = .75 W/RTT
  - **Good formula to remember**
  - Required window size W = 83,333 in flight segments
- Magic formula to relate loss rate to throughput
  - Throughput = \((1.22 * \text{MSS})/(\text{RTT} \cdot \sqrt{\text{L}})\)
  - In this example, we can only lose one in 5,000,000,000 segments. If we lose more than this, it is bad
  - SO, we must adapt TCP to work in fast high speed links

TCP Fairness:
- Over a large amount of time, does TCP behave in a fair manner?
- EX: we have a single router with two connections. Assume connections have the same segment size and RTT. We assume that over a long period of time, both connections will get R/2
- Goal: if K TCP sessions share same bottleneck link of bandwidth R, each session will get R/K capacity
- Two competing sessions (see graph in slides)
  - Additive increase gives slope of 1, as throughput increases
  - Multiplicative decrease drops throughput proportionally
- In theory, the connections will become fair over time by decreasing window time when there is congestion and packet loss
- UDP does not care about fairness
  - No condition control
  - Push data as fast as you can
  - Use UDP for applications when you don’t want congestion to affect anything
- TCP parallel connections
  - App can open parallel connections between 2 hosts
  - Web browsers do this
  - This allows the app to have better bandwidth

Queuing Mechanisms
- TCP relies on dropped packets to detect congestion
  - TCP drives the network into packet loss by continuing to increase the sending rate
- Drop-Tail queuing leads to bursty loss
  - When a link becomes congested, many arriving packets encounter a full queue
  - So many flows divide sending rate in half and man individual flows lose multiple packets

Random Early Detection (RED)
- Once the router notices that the queue is getting filled up, it will mark random queued packets to drop to signal congestions
- This way we still have room in the queue for incoming packets
- Packet drop probability
  - If queue length less than threshold, probability of dropping packet is low
  - If queue length is more than max threshold, probability of dropping packet is 1
  - Drop probability increases as queue length increases
- Properties of RED
  - Drop packets before queue is full to reduce the rates of soe flows.
    - This adds some fairness packets are dropped in a fair manner
  - Drop packets in proportion to each flow’s rate
  - Drops are spaced out in time
  - Tolerant of burstiness in the traffic
- Problems with RED
  - Hard to get the tunable parameters just right because it relies on experimentation for packet dropping
    - How early to start dropping packets?
    - What slope for the increase in drop probability?
  - Sometimes RED helps but sometimes not
  - RED is implemented in practice, but often not used due to challenges of tuning right.
Explicit Congestion Notification
- Early dropping of packets
  - Good: early feedback
  - Bad: you have to drop the packet to give feedback
- Router marks the packet with an ECN bit and sending host interprets as a sign of congestion
- Challenges
  - Must be supported by the end hosts and the routers (not always the case, people don’t always have the same implementation of TCP)

Conclusions
- Congestion in internet is inevitable
- Congestion can be handled
  - Active queue management can help

End TCP