Administrivia

- New project online due Wednesday of final’s week (hard deadline)

- Options
  1. Standard option: do project 3 basic part, take final
  2. Exams are evil option: do extra bonus parts to get out of final
  3. I hate C++/C and programming option: Term paper for 85% of credit of project 3 – talk to me for a topic

- Last time: TCP details and Midterm review
Sliding Window Operation in TCP

- TCP’s sliding window is a hybrid of Selective repeat and Go-Back-N
  - Like selective repeat, buffer segments that arrive out of order
  - Like Go-Back-N, uses cumulative ACKs

- Sender window size obtained by explicit feedback
  - What size is it?
  - Later, we will update to incorporate congestion
Flow Control

- Advertised window is: \((\text{MaxRcvBuffer} - (\text{LastByteRcvd} - \text{LastByteRead}))\)

- Sender sends if the packets it has already sent out are less than the advertised window
  - Effective Send Window = Advertised Window - (Last Byte Sent - Last Byte ACK’d)
  - Why this conservative?
Big Fat Pipe Extension

- Problem: TCP sequence number field wraparound
- Problem: TCP advertised window limitation
- One more problem with TCP is the granularity of the timer (originally 0.5 seconds)
- Extension addresses wraparound as well as timer granularity
- Implemented using the optional portion of the TCP header
- Stores timestamps in outgoing segments, timestamp included in reply
  - System time can be used to provide fine grained estimates
  - Timestamp effectively extends the sequence number solving the wraparound problem
- Extension also allows the peers to agree on a scaling factor for the advertised window
Revisiting Sliding Window

● Problem: End-to-end delay is highly variable
  – All kinds of different connections – TCP connection on the same subnet to a connection across the world
  – With other traffic on the network, the same connection varies over time

● What should the timeout value be?
Adaptive Retransmission

- Idea: Make it a function of the measured RTT

- How to measure RTT?
  - Difference between send time and ACK receipt?
  - What about delayed ACKs?
  - Timer granularity hurts

- What to do once we measure RTT?
Original Algorithm

- Measure RTT for every segment/ACK (with delayed ACKs every other segment)

- Keep weighted average of RTT as:
  - Estimated RTT = $\alpha$ EstimatedRTT + $(1-\alpha)$ SampleRTT
  - $\alpha$ between 0.8 and 0.9
  - RTO = 2 x EstimatedRTT

- What do you think?
Problem

- Flaw with the original algorithm: what happens on a retransmission?
  - If we assume ACK is for the original packet, you could overestimate RTT
  - If we assume ACK is for the retransmission, we can underestimate
- Does it matter with timestamp extension??
Karn/Partridge Extension

- Update EstimatedRTT (also called Smoothed RTT or SRTT) only on segments that have been sent once

- Introduce exponential backoff of Timeout on each retransmission
  - Similar to backoff in MAC layer; but no collisions here, why is it necessary??
  - Think about congestion
Jacobson/Karels Algorithm

- Idea: make timeout reflect variance – not only average RTT

- Algorithm (note, following original paper, not book format):
  
  - Estimate RTT and RTT “variance” as follows
    \[ \text{EstimatedRTT} = (1 - \alpha)\text{EstimatedRTT} + \alpha \text{SampleRTT} \]
    \[ \text{Diff} = |\text{SampleRTT} - \text{EstimatedRTT}| \]
    \[ \text{Deviation} = (1 - \beta)\text{Deviation} + \beta(\text{Diff}) \]

- Set timeout to be
  \[ \text{EstimatedRTT} + \phi\text{Deviation} \]

- Typical values for \( \alpha \), \( \beta \) and \( \phi \) are 1/8 and 1/4 and 4 respectively

- What is the intuition, is this better than Karn/Partridge?
• Problem: how to prevent a dynamic set of users who are not aware of each other from overflowing the shared network resources

• Related problem: if the demand exceeds the capacity, how do you share the network fairly (or in proportion to promised service)?

• Think of collisions in a shared medium: a simple form of congestion
General Approaches

- Ignore it until there is a problem then take action to control it (conceptual equivalent of contention-based MAC)

- Pre-allocate resources to avoid problems (conceptual equivalent of what?)
TCP Congestion Control

- Problem (noticed by van Jacobson in 1988)
  - Hosts sending as fast as they are allowed
  - Congestion happens, packets are dropped
  - Hosts timeout and retransmit packets, increasing congestion

- TCP Congestion Control:
  - Assumes a best-effort network (router queueing is an independent mechanism)
  - Host-based solution: the TCP ends try to figure out what is happening based on the behavior of the network
  - ACKs pace transmission (self-clocking)

- Challenges:
  - How to determine the end to end capacity at the beginning
  - How to adapt to dynamic changes
Implementation: Congestion Window

- Objective: adjust to changes in the available capacity
- Add a new variable per TCP connection: CongestionWindow
  - Limits how much data the source has in transit (how is this different from advertised window)?

\[
\text{MaxWindow} = \min(\text{CongestionWindow}, \text{AdvertisedWindow})
\]
\[
\text{EffectiveWindow} = \text{MaxWindow} - (\text{LastByteSent} - \text{LastByteAcked})
\]
- How should this window size be set and adapted? How about at setup time?
Additive Increase/Multiplicative Decrease (AIMD)

- Idea, find it adaptively:
  - When no congestion, slowly increase window (what if it becomes larger than Advertised Window?)
  - When congestion occurs, reduce window quickly
  - Must cause congestion in order to find what the appropriate rate is

- How do you know when congestion occurs?
  - When a timeout happens, usually means a packet is lost due to congestion
  - What if the packet was lost due to an error?

- Why not additive increase/additive decrease?
Implementation

- Every time a “CongestionWindow” worth of data has been successfully sent, increment the window by one packet (Additive Increase)
- Everytime a timeout occurs, divide the congestion window by 2 (multiplicative decrease)
- In practice, increment a little for each ACK
  \[
  \text{Increment} = \text{MSS} \cdot \frac{\text{MSS}}{\text{CongestionWindow}}\]
  \[
  \text{If (ACKRecvd)}
  \]
  \[
  \text{CongestionWindow}+ = \text{Increment}
  \]
Infamous TCP Sawtooth

- Congestion Control mechanism in action: oscillating around the optimal sending rate – can we do better?
- Problem: Additive increase is too slow to get to capacity when the connection starts
  - How about fast start – send at the maximum possible local rate?
• Idea: quickly find the initial capacity of the network

• Implementation:
  – Begin with CongestionWindow of 1 packet (MSS)
  – Double CongestionWindow each RTT (increment by 1 packet for each ACK)
• Problem: can send up to double the appropriate congestion level
  – Why do we have to do this?
  – Solution: remember the last successful send window size (threshold)
  – Go to additive increase next time it is reached

• More clever solutions? Packet pair
  – Send two packets simultaneously and check spacing on the ACKs

• Problem: timer granularity is too big
Problem: coarse-grained TCP timeouts

Fast retransmit:
- If we see duplicate acks for the same segment, it is likely the segment after it was lost
- Resend it without the timer expiring

What should be done when a packet is retransmitted? Fast recovery (TCP-Reno)
- Half congestion window value when a segment is retransmitted (rather than going to slow start with the updated threshold)
Aside – TCP versions

- Original RFC 793 (original adaptive RTO algorithm)
- RFC 1122: RTT variance estimation, Karn’s algorithm, Slow start, Dynamic Window sizing on congestion
- TCP Tahoe (1988; BSD NR 1.0) is the original BSD TCP released: includes everything in RFC 1122 + Fast retransmission
- TCP Reno (BSD NR 2.0) added Fast recovery and delayed Acks (every other segment); newer versions include the long fat network extension
- New Reno; experimental – packet pair, Hoe’s partial ACK to stay in fast recovery
- SACK extension: signal multiple lost segments together (standardized but optional)
- TCP Vegas is an experimental extension that incorporates Congestion Avoidance
• Better performance with fast retransmit – some timeouts are avoided
  – If not enough packets in the advertised window, might not be able to detect loss.
  – Also if losses are too big, might not get a triplicate ACK
  – If two segments in a row get lost, need 6 acks, etc...

• Still need to cause congestion to control congestion – can we avoid congestion altogether?
• Source based congestion avoidance

• Intuition: source watches for signs of congestions

• What are possible indicators of congestion that is about to happen but has not happened yet?
Anatomy of Congestion

- When we are operating above the capacity of the network
  - Buffers at intermediate routers will start to be reserved
  - RTT will increase (queueing delays)
  - Even though we are sending at higher than the network capacity, the network will deliver only what it can

- Possible congestion warnings:
  - RTT increases
  - Observed throughput is lower than expected throughput
TCP Vegas

- Let BaseRTT be the minimum measured RTT

- Notice that if congestion window exceeding the connection’s capacity then
  - Expected throughput = CongestionWindow/BaseRTT

- TCP Vegas Congestion Avoidance:
  - Calculate sending rate (Actual Rate) once per RTT
  - Let Diff be Expected Rate - Actual Rate
    1. If Diff is less than $\alpha$ increase congestion window linearly
    2. If Diff is more than $\beta$ reduce congestion window linearly
    3. Otherwise leave window unchanged
  - $\alpha < \beta$ (typically 1 and 3 “packets”)
  - What is the intuition?
**Implementation completely at the source – very good**
Other Improvements in TCP Vegas

- Improve Retransmission
  - Timestamp outgoing segments
  - Check if timeout expired on receiving the first duplicate ack and retransmit
  - Check the next few segments after a retransmission to see if another retransmission is necessary

- Modify slow start to avoid congestion as well
  - Instead of doubling every RTT, double every other RTT
  - Keep RTT the same for one window after doubling – check congestion ($\alpha\beta$ test as above)
  - If congestion, go to additive increase