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
- Extension also allows the peers to agree on a scaling factor for the advertised window
- System time can be used to provide fine-grained estimates of sequence number
- Stores timestamps in outgoing segments, timestamp included in reply
- Time stamps effectively extend the sequence number granularity
- extension addresses wraparound as well as timer granularity
- One more problem: TCP advertised window limitation

Administivia

- 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

- Like selective repeat, buffer segments that arrive out of order
- Latest, we will update to incorporate congestion
- What size is it?
- Extension also allows the peers to agree on a scaling factor for the advertised window
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?

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?

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

**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?)

**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
  - EstimateRTT = \( \beta \) \* EstimatedRTT + \((1-\beta)\)Diff
  - Diff = SampleRTT - EstimatedRTT
  - \( \beta \) typical values for \( \alpha, \beta, \text{ and } \phi \) are 1/8 and 1/4 and 4 respectively

- Set timeout to be
  - EstimatedRTT + Deviation

- What is the intuition, is this better than Karn/Partridge?
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?

SUNY-BINGHAMTON – CS428 SPRING ’05 LEC. #19

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/multiplicative decrease?

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)?

  $\text{MaxWindow} = \text{Min}(\text{CongestionWindow, AdvertisedWindow})$

  $\text{EffectiveWindow} = \text{MaxWindow} - (\text{LastByteSent} - \text{LastByteAcked})$

- How should this window size be set and adapted? How about at setup time?
**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?**

**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

---

**Slow Start**

- **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)

**Fast Retransmit/Recovery**

- **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

Discussion

- 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?

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:
  - Observed throughput is lower than expected throughput
  - RIT increases
  - RIT will increase (queueing delays)
Let $BaseRTT$ be the minimum measured RTT.

Notice that if congestion window exceeding the connection's capacity then:

1. **TCP Vegas Congestion Avoidance**
   - Calculate sending rate (Actual Rate) per connection.
   - If $Diff$ is more than $\Delta$ reduce congestion
   - If $Diff$ is less than $\Delta$ increase congestion
   - If $Diff$ is equal to Expected Rate - Actual Rate

2. Check the next few segments after a retransmission to see if another retransmission is necessary
   - Keep RTT the same for one window after doubling - check congestion as above
   - If congestion, go to additive increase
   - If $Diff$ is less than $\Delta$ increase congestion
   - If $Diff$ is more than $\Delta$ reduce congestion
   - Otherwise keep window unchanged

3. **Other Improvements in TCP Vegas**
   - Improve Retransmission
     - Timestamp outgoing segments
     - Check if timeout expired on receiving the first duplicate ack and retransmit
     - Check if timeout expired on receiving the first duplicate ack and retransmit
     - Instead of doubling every RTT, double every other RTT

Other improvements include:

- 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 as above
- If congestion, go to additive increase

**TCP Vegas Operation**

- Expected throughput = $\frac{CongestionWindow \times BaseRTT}{2}$
- Notice that if congestion window exceeding the connection's capacity then:
  - Let $BaseRTT$ be the minimum measured RTT