EECS 122, Lecture 22

Today's Topics:

TCP Congestion Control

Fast Retransmit

Round-Trip Estimation & Time-out

Silly Window Syndrome

Kevin Fall, kfall@cs.berkeley.edu

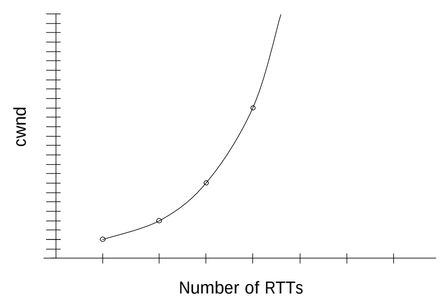
TCP Slow Start

- Slow-start is a TCP behavior used to get to packet equilibrium
- Slow-start increases the congestion window *exponentially*, rather than linearly
- Why called slow-start then?
  - well, it is considerably slower than what used to happen (start based only on the receiver's advertised window)

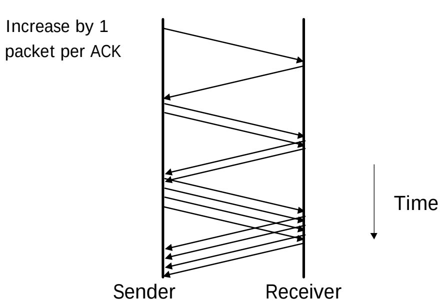
TCP Slow Start

- For each ACK received, increase the congestion window by 1
- Results in cwnd pattern of: 1, 2, 4, 8, 16, 32, …
  - takes time proportional to $\log_2 W$ to reach window of W, [longer if ACKs delayed]

TCP Slow Start



TCP Slow Start



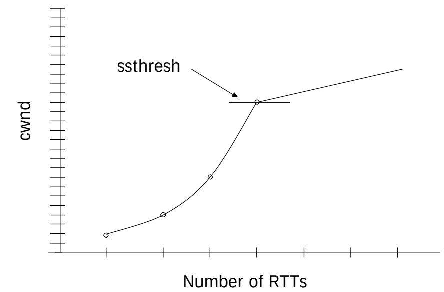
TCP Congestion Behaviors

- Two algorithms:
  - slow-start: getting to equilibrium
  - congestion avoidance: searching for new available bandwidth in path (and reacting to congestion)
- The two behaviors are mutually exclusive for any single point in time, but each TCP implements both:
  - establish an operating point to switch between the two algorithms (*ssthresh*)

Slow-Start Threshold (ssthresh)

- Need a way to determine whether the TCP should do slow-start or congestion avoidance
- New variable (ssthresh):
  - if cwnd <= ssthresh, do slow-start
  - if cwnd > ssthresh, do congestion avoidance
- ssthresh is initialized to a large value, after a congestion signal, cwnd is divided in half, and ssthresh is set to cwnd

TCP Slow-Start & Congestion Avoidance



ssthresh and cwnd maintenance

- Congestion window is normally divided on congestion indications (packet dops), and grows linearly if above ssthresh
- ssthresh is reset to cwnd after it is reduced to keep a marker of the last operating point
- so, when do we ever enter slow-start after a connection has started?

Detecting Loss with TCP

- TCP uses lost packets as indicators of congestion
- Two methods
  - timer expiring
  - fast retransmit
- Fast retransmit:
  - because of cumulative ACK, out-of-order data received at receiver may generate *duplicate ACKs* (“dupacks”)

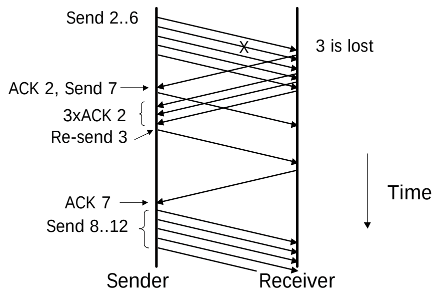
Duplicate ACKs

- We arrange for TCPs receiving out-of-order packets to respond immediately with one ACK per packet:
  - receiver gets: 5, 6, 7, 8, 10, 11, 12, 13
  - ACKs: 6, 8, 8, 8, 8, 8 [4 dupacks]
- Provides a hint to sender that packet 9 is probably missing at receiver and that 4 packets have arrived after 8 arrived
- [think about re-ordering!]

Fast Retransmit

- Heuristic at sender to trigger retransmissions w/out timeouts
- To avoid retransmitting due to small re-ordering, look for 3 DUPACKS
- So, on 3rd dupack for packet n, retransmit n+1, and send more if send window allows
- If only one packet lost, fills receiver’s “hole”, resulting in ACK for top of window

Fast Retransmit Example



Fast RTX Observations

- Fast retransmit can repair modest packet lost without requiring a retransmission timer to expire
- Because it requires 3 dupacks to fire, doesn't work so well with small windows (because there won't be enough ACKs generated at the receiver)
- With large numbers of dropped packets, similar problem (not enough ACKs)

Congestion Action on Loss

- TCP has different behaviors, depending on the way it detects loss (RFC2001):
  - RTX timer expires:
    - ssthresh = MAX(MIN(win,cwnd)/2,2)
    - cwnd = 1 (initiates slow-start)
  - fast retransmit (fast recovery):
    - ssthresh = MAX(MIN(win,cwnd)/2,2)
    - cwnd = ssthresh + 3
    - each additional dupack increments cwnd by 1
      - fast recovery
      - (cwnd = ssthresh on new ACK)

TCP Congestion Behavior (summary)

- Slow-start:
  - new connection, after idle time, after RTX timer expires
  - set cwnd=1, grow window exponentially
  - searches quickly for operating point
- Congestion avoidance:
  - normal operations, fast RTX/recovery
  - divide operating point in 1/2 after loss
  - searches slowly for new bandwidth

Setting TCP's RTX Timers

- Slow-start is invoked as a result of a timer expiring (resetting the world)
- Recall we need some way of setting this timer, but TCP must work both in local as well as very long delay environments
- Need a way to set the timer based on the connection's round-trip time:
  - how to measure the RTT?
  - how to set the RTX timer based on this?

Measuring the RTT

- Should be very simple:
  - when sending a packet, jot down the time
  - when receive the ACK for it, take the difference and call that the RTT
- Problem:
  - in TCP, no way to tell whether an ACK was for an original or retransmitted packet
  - called "acknowledgement ambiguity"

Karn's Algorithm

- Really two parts…
- To solve ACK ambiguity:
  - do not measure the RTT for segments that have been retransmitted (simple)
- On a timeout:
  - network is telling you it is having trouble
  - so, double RTX timer (up to 64x) on each subsequent timeout (64s max)

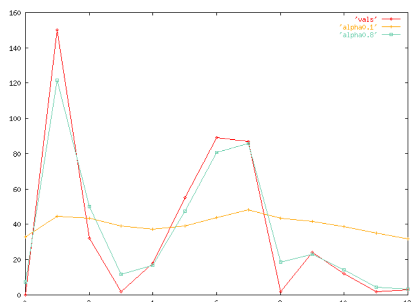
Estimating the RTT

- To estimate the connection's round-trip time, TCP uses an exponentially weighted moving average (like RED):

$$W_t = \alpha m_t + (1-\alpha)W_{t-1}$$

- Also called a low-pass filter
- Requires only 1 word of memory

EWMA Example



Properties of the EWMA

- Also sometimes expressed as:

$$W_t = \alpha(m_t - W_{t-1}) + W_{t-1}$$

- This form is useful because it involves only one multiply (computationally expensive as compared with add or subtract)

TCP RTT Measurement

- Early TCPs used just the mean RTT estimate and set the timer to be 2x this estimate…the 2 accounting for some amount of variance
- In large-variance networks, though, this might not be enough. How to measure the variability of the RTT as well…?
- Perhaps the standard deviation…

Measuring Variability

- Most common measure of sample variability is sample variance S^2 [square of the standard deviation]:

$$S^2 = \frac{\sum_{i=1}^{n} (m_i - \overline{X})^2}{n-1}$$

- Not very efficient for a protocol implementation due to the square root needed to get the sample std. deviation

Measuring Variability

- Alternative is to use the *mean deviation* (or *mean absolute deviation--MAD*):

$$MD = \frac{\sum_{i=1}^{n} |m_i - \bar{X}|}{n}$$

- No need to square or take square root. Units are same as mean. Not commonly used because of less nice predictive properties than standard deviation.

Setting the TCP RTX Timeout

- TCP uses a combination of the mean and mean deviation estimators:
  - RTT = (1-g)*RTT + g * [rtt sample]
  - D = (1-h)*D + h * |sample - RTT|
  - g = 0.125 (2^-3), h = 0.25 (2^-2)
  - efficiently implemented using fixed point arithmetic
- So, 95% of the time would expect:
  - (RTT-2D)<(actual RTT)<(RTT+2D) if normal

Setting the TCP RTX Timeout

- But RTTs don't seem to be Gaussian, so additional "fuzz" is used:
  - RTO = RTT + 4 * D
- In addition, many TCPs use an imprecise clock that only "ticks" every 500ms. All RTT measurements (and timeouts) use this tick rate.
- Only a single timer maintained usually

Silly Window Syndrome

- Recall TCP is a window-based protocol
- What happens if a receiver with a small buffer advertises it, and sender quickly fills it with a small amount of data?
  - inefficient use of bandwidth by sending high-overhead "tinygrams"
- What to do?
  - want a way to "save up" enough to send, and do so only when "worth it"

Nagle's Algorithm

- Purpose is to avoid inefficient use of bandwidth
- Sender operation:
  - buffer all user data if any unacknowledged data is outstanding
  - ok to send if all ACKd or have a full packet (MSS) size worth of data to send
- Receiver operation
  - ok to send if can open recv window enough

Receive Side SWS Avoidance

- Receiver resists advertising a window bigger than it is currently advertising (which might be zero) unless it can be increased by at least

  MIN(one MSS, 0.5 * receiver's available buffer)

- Same bit of logic ensures that window shrinkage does not occur

Properties of Nagle Algorithm

- Applies only to small packets. For bulk data transfers, always have a full MSS to send
- Algorithm is self-clocking:
  - basically does Stop&Wait for small packets
  - on LAN, small RTT implies not much wait, but inefficient
  - on WAN, large implies more wait, but more efficient on long links [where it counts most]

Impact of Nagle Algorithm

- When small delay is needed, Nagle algorithm can cause unwanted packet delays
- Applications can disable this algorithm:

```
int one = 1;
setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, &one,
  sizeof(one))
```

Where we are so far with TCP

- Important algorithms
  - congestion avoidance
  - slow start
  - round-trip time estimation
  - Karn's timer backoff
  - silly window avoidance/Nagle
- We don't yet know about connection establishment (next time…)