TCP Timeout & Congestion Control

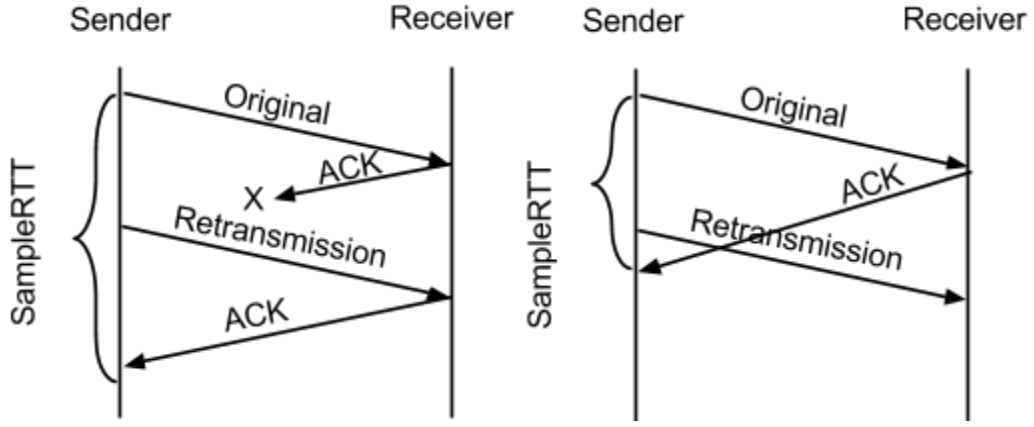
CS640, 2015-03-17

Outline

- Setting timeouts
- Congestion control overview
- Exponential backoff
- Slow start

Setting Timeouts

- Ideal -- RTO = RTT
  - RTT can vary over time
  - ***How do we estimate RTT?**
    - Sending special probe packets (e.g., ICMP ping) -- introduces overhead
    - Based on data packets and ACKs
- Basic method
  - Record time when TCP sends data segment
  - Record time when TCP receives ACK for the segment
  - Difference in times is SampleRTT
  - Compute estimated weighted moving average (EWMA) of RTT
    - EstimatedRTT = α * EstimatedRTT + (1 - α) * SampleRTT
    - $0.8 < \alpha < 0.9$
  - Conservatively set RTO based on EstimatedRTT
    - RTO = 2 * EstimatedRTT
    - Preferable to add extra delay instead of extra traffic, because loss usually occurs when network is congested and more traffic will exacerbate the problem
- ***Does anyone see a problem?**
  - When data is retransmitted, we don't know if ACK is for original transmission or retransmission



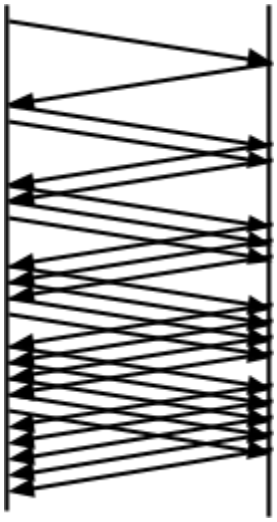
- Karn/Partridge algorithm
    - Don't sample RTT when data is retransmitted -- only measure for data that is sent once
    - Exponential backoff of RTO
        - Every time data is retransmitted, RTO is set to twice the previous RTO
        - Intuition -- Timeout occurs when data or ACK packets are lost, which usually happens when there is congestion; want a conservative RTO to avoid exacerbating congestion through spurious retransmissions
        - Problem -- too conservative
- Jacobson/Karels algorithm
    - Basic method for setting RTO does not consider variance of RTT samples
    - If we know the variance, then we can better set RTO
        - Difference = SampleRTT - EstimatedRTT
        - EstimatedRTT = EstimatedRTT + (**δ** * Difference)
        - Deviation = Deviation + **δ**(|Difference| - Deviation)
        - 0 < **δ** < 1
        - RTO = $\boldsymbol{\mu}$ * EstimatedRTT + $\boldsymbol{\phi}$ * Deviation
        - $\boldsymbol{\mu}$ = 1
        - $\boldsymbol{\phi}$ = 4
    - When variance is high, amount of deviation will dominate the calculated RTO; when variance is low, RTT will dominate the calculated RTO

TCP Congestion Control

- Congestion control introduced in 1989 by Van Jacobson
    - About 8 years after TCP and IP were used in the Internet
    - Internet was suffering from congestion collapse
        - Hosts sent as much data as flow control (i.e., advertised window) allowed
        - Causing congestion at routers -> packet loss
        - Retransmission of lost packets caused more congestion
- Goal: make TCP aware of congestion and react in a way that controls the problem
    - Need to estimate how much capacity is available in the network
    - Challenge: other connections start/end, and change their packet rate due to flow control
- Mechanisms
    - Exponential backoff
    - Slow start
    - Fast retransmit and fast recovery
    - Congestion avoidance

Exponential Backoff

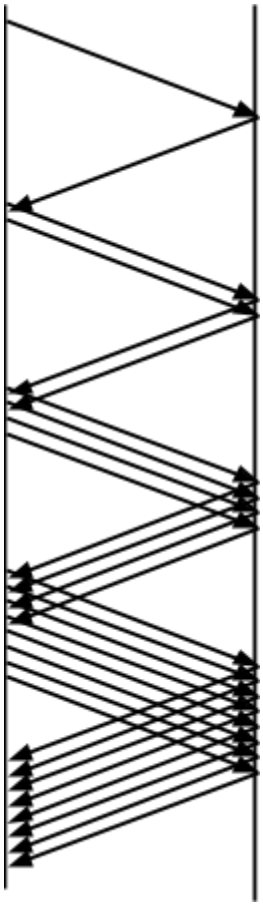
- Introduce new variable: Congestion Window (CWND)
    - Limits amount of data in flight (i.e., sent but unACKed data) based on network conditions
    - MaxWin = min(CWND, RWND)
      CWND = CongestionWindow
      RWDN = AdvertisedWindow
    - Now: EffectiveWindow = MaxWindow - (LastByteSent - LastByteAck'd)
- How is Congestion Window set? -- based on amount of congestion in the network
- ****How do we know there is congestion?***
    - Treat loss as a sign of congestion
    - Timeouts indicate loss
    - Loss could also be due to errors, but this is "rare"
        - Except in wireless networks and physical networks with poor connections
        - Dale will talk about challenges of new physical mediums on Thursday
- CWND is defined in bytes, but it's easier to think about congestion in terms of # of MSS-byte packets
- Additive Increase/Multiplicative Decrease (AIMD)
    - Start CWND = 1 pkt
    - Increase CWND by 1 pkt every RTT -- CWND += 1
        - In practice, increment CWND by a fraction for every received ACK
          CWND = CWND + (MSS * (MSS / CWND))
    - On timeout, divide CWND in half -- CWND = CWND / 2 (never go below 1 pkt)
    - Decrease aggressively and increase conservatively because having too large a window is much worse than having too small a window
        - More drops => more retransmissions => more congestion
    - Timeline (Figure 6.8 on pg. 502)



    - Results in sawtooth behavior
- ****What is a problem with this approach?***
    - Takes a long time to reach maximum bandwidth
        - About 83 RTTs to reach maximum capacity of 1Mbps link
        - Modern day links are 10-30Mbps (home internet), 1Gbps (LAN), 10Gbps (data center)
    - Want to discovery path capacity more quickly

Slow Start

- Introduce new variable: Slow Start Threshold (SSTHRESH)
    - Exponentially increase CWND while CWND < SSTHRESH
- Algorithm
    - Begin with CWND = 1, SSTHRESH = infinity
    - Double CWND every RTT -- CWND = CWND * 2
        - In practice, add 1 to CWND for every received ACK -- CWND += 1
    - Switch to additive increase (AI) when CWND >= SSTHRESH
    - On loss, set SSTHRESH = CWND/2 (keep SSTHRESH >= 1) and CWND = 1
        - Multiplicative decrease of CWND is used to set SSTHRESH, but CWND itself is set to 1
    - Timeline (Figure 6.10 on pg 506)



    - Graph of behavior (Figure 6.11 on pg. 508)
- Lots of loss can occur during initial slow start phase
    - Assume network capacity is 16
    - Assume current value of CWND is 16, but all ACKs are received so we set CWND = 2 * 32
    - Now half of pkts (16 of 32) will be dropped -- this is the worst case
    - If network capacity where 31, then 1 pkt would be dropped -- this is the best case
    - Problem becomes increasingly severe as delay x bandwidth product increases
        - RTT of 30ms (delay to nytimes.com)
        - Bandwidth of 20Mbps (reasonably high-speed home Internet)
        - delay x bandwidth product = 75KB
        - Could lose up to 75KB of data (more than 50 pkts) during initial slow start phase