Introduction to Computer Networks

COSC 4377

Lecture 9

Spring 2012
February 15, 2012
Announcements

• HW4 due today
• Start working on HW5
• In-class student presentations
• TA office hours this week
  – TR 1030a – 100p
Today’s Topics

• HW4 and HW5 discussions
• Transport Protocols
  – Flow Control
  – Congestion Control
HW4 and HW5

• Web Server Concurrency
  – Fork vs Thread vs Select
• Use large file for testing
Flow Control

• Goal: not send more data than the receiver can handle
• Sliding window protocol
• Receiver uses window header field to tell sender how much space it has
Congestion Control

• Goal: do not send more data than the network can take

• 3 Key Challenges
  – Determining the available capacity
  – Adjusting to changes in the available capacity
  – Sharing capacity between flows
Slow Start

Figure 3: Startup behavior of TCP without Slow-start

Figure 4: Startup behavior of TCP with Slow-start

From [Jacobson88]
Slow start implementation

• Let $w$ be the size of the window in bytes
  – We have $w$/MSS segments per RTT
• We are doubling $w$ after each RTT
  – We receive $w$/MSS ACKs each RTT
  – So we can set $w = w + $MSS on every ack
• At some point we hit the network limit.
  – Experience loss
  – We are at most one window size above the limit
  – Remember this: ssthresh and reduce window
Slow Start

- We double cwnd every round trip
- We are still sending min (cwnd,rcvwnd) pkts
- Continue until ssthresh estimate or pkt drop

http://en.wikipedia.org/wiki/Slow-start
Dealing with Congestion

• Assume losses are due to congestion
• After a loss, reduce congestion window
  – How much to reduce?
• Idea: conservation of packets at equilibrium
  – Want to keep roughly the same number of packets network
  – Analogy with water in fixed-size pipe
  – Put new packet into network when one exits
How much to reduce window?

• What happens under congestion?
  – Exponential increase in congestion

• Sources must decrease offered rate exponentially
  – i.e, multiplicative decrease in window size
  – TCP chooses to cut window in half
How to use extra capacity?

• Network signals congestion, but says nothing of underutilization
  – Senders constantly try to send faster, see if it works
  – So, increase window if no losses... By how much?

• Multiplicative increase?
  – Easier to saturate the network than to recover
  – Too fast, will lead to saturation, wild fluctuations

• Additive increase?
  – Won’t saturate the network
Chiu Jain Phase Plots

Fair: $A = B$

Efficient: $A + B = C$

Goal: fair and efficient!
Chiu Jain Phase Plots

Flow Rate A

Flow Rate B

Fair: $A = B$

Efficient: $A + B = C$

MIMD
Chiu Jain Phase Plots

Fair: \( A = B \)

Efficient: \( A + B = C \)

Flow Rate A

Flow Rate B
Chiu Jain Phase Plots

- Fair: $A = B$
- Efficient: $A + B = C$

Flow Rate B

Flow Rate A

AIMD
AIMD Implementation

• In practice, send MSS-sized segments
  – Let window size in bytes be \( w \) (a multiple of MSS)
• Increase:
  – After \( w \) bytes ACKed, could set \( w = w + \text{MSS} \)
  – Smoother to increment on each ACK
    • \( w = w + \text{MSS} \times \frac{\text{MSS}}{w} \)
    • (receive \( w/\text{MSS} \) ACKs per RTT, increase by \( \frac{\text{MSS}}{\left(\frac{w}{\text{MSS}}\right)} \) for each)
• Decrease:
  – After a packet loss, \( w = w/2 \)
  – But don’t want \( w < \text{MSS} \)
  – So react differently to multiple consecutive losses
  – Back off exponentially (pause with no packets in flight)
AIMD Trace

- AIMD produces sawtooth pattern of window size
  - Always probing available bandwidth
Putting it together

• TCP has two states: Slow Start (SS) and Congestion Avoidance (CA)
• A window size threshold governs the state transition
  – Window <= threshold: SS
  – Window > threshold: congestion avoidance
• States differ in how they respond to ACKs
  – Slow start: \( w = w + \text{MSS} \)
  – Congestion Avoidance: \( w = w + \frac{\text{MSS}^2}{w} \) (1 MSS per RTT)
• On loss event: set \( w = 1 \), slow start
How to Detect Loss

• Timeout

• Any other way?
  – Gap in sequence numbers at receiver
  – Receiver uses cumulative ACKs: drops => duplicate ACKs

• 3 Duplicate ACKs considered loss
Putting it all together
RTT

• We want an estimate of RTT so we can know a packet was likely lost, and not just delayed
• Key for correct operation
• Challenge: RTT can be highly variable
  – Both at long and short time scales!
• Both average and variance increase a lot with load
• Solution
  – Use exponentially weighted moving average (EWMA)
  – Estimate deviation as well as expected value
  – Assume packet is lost when time is well beyond reasonable deviation
Originally

- \( \text{EstRTT} = (1 - \alpha) \times \text{EstRTT} + \alpha \times \text{SampleRTT} \)
- \( \text{Timeout} = 2 \times \text{EstRTT} \)
- Problem 1:
  - in case of retransmission, ack corresponds to which send?
  - Solution: only sample for segments with no retransmission
- Problem 2:
  - does not take variance into account: too aggressive when there is more load!
Jacobson/Karels Algorithm (Tahoe)

- \( \text{EstRTT} = (1 - \alpha) \times \text{EstRTT} + \alpha \times \text{SampleRTT} \)
  - Recommended \( \alpha \) is 0.125
- \( \text{DevRTT} = (1 - \beta) \times \text{DevRTT} + \beta \mid \text{SampleRTT} - \text{EstRTT} \mid \)
  - Recommended \( \beta \) is 0.25
- Timeout = \( \text{EstRTT} + 4 \times \text{DevRTT} \)
- For successive retransmissions: use exponential backoff
Old RTT Estimation

Figure 5: Performance of an RFC793 retransmit timer

Packet RTT (sec.)

Trace data showing per-packet round trip time on a well-behaved Arpanet connection. The x-axis is the packet number (packets were numbered sequentially, starting with one) and the y-axis is the elapsed time from the send of the packet to the sender’s receipt of its ack. During this portion of the trace, no packets were dropped or retransmitted. The packets are indicated by a dot. A dashed line connects them to make the sequence easier to follow. The solid line shows the behavior of a retransmit timer computed according to the rules of RFC793.

The parameter $\tau$ accounts for RTT variation (see [5], section 5). The suggested $\tau = 2$ can adapt to loads of at most 30%. Above this point, a connection will respond to load increases by retransmitting packets that have only been delayed in transit. This forces the network to do useless work, wasting bandwidth on duplicates of packets that will eventually be delivered, at a time when it's known to be having trouble with useful work. I.e., this is the network equivalent of pouring gasoline on a fire.

We developed a cheap method for estimating variation (see appendix A) and the resulting retransmit timer essentially eliminates spurious retransmissions. A pleasant side effect of estimating $\tau$ rather than using a fixed value is that low load as well as high load performance improves, particularly over high delay paths such as satellite links (figures 5 and 6).

Another timer mistake is in the backoff after a retransmit: If a packet has to be retransmitted more than once, how should the retransmits be spaced? For a transport endpoint embedded in a network of unknown topology and with an unknown, unknowable and constantly changing population of competing conversations, only one scheme has any hope of working—exponential backoff—but a proof of this is beyond the scope of this paper.

We are far from the first to recognize that transport needs to estimate both mean and variation. See, for example, [6]. But we do think our estimator is simpler than most.

See [8]. Several authors have shown that backoffs ‘slower’ than exponential are stable given finite populations and knowledge of the global traffic. However, [17] shows that nothing slower than exponential behavior will work in the general case. To feed your intuition, consider that an IP gateway has essentially the same behavior as the ‘ether’ in an ALOHA net or Ethernet. Justifying exponential retransmit backoff is the same as...
To finesse a proof, note that a network is, to a very good approximation, a linear system. That is, it is composed of elements that behave like linear operators — integrators, delays, gain stages, etc. Linear system theory says that if a system is stable, the stability is exponential. This suggests that an unstable system (a network subject to random load shocks and prone to congestive collapse) can be stabilized by adding exponential damping (exponential timer backoff) to its primary excitation (senders, traffic sources).

If the timers are in good shape, it is possible to state with some confidence that a timeout indicates a lost packet and not a broken timer. At this point, something can be done about (3). Packets get lost for two reasons: they are damaged in transit, or the network is congested and somewhere on the path there was insufficient buffer capacity. On most network paths, loss due to damage is rare (≪1%) so it is probable that a packet loss is due to congestion in the network.

Because a packet loss empties the window, the throughput of any window flow control protocol is quite sensitive to damage loss. For an RFC793 standard TCP running with window \( w \) (where \( w \) is at most the bandwidth-delay product), a loss probability of \( p \) degrades throughput by a factor of \( (1 + 2pw)^{-1} \). E.g., a 1% damage loss rate on an Arpanet path (8 packet window) degrades TCP throughput by 14%.

The congestion control scheme we propose is insensitive to damage loss until the loss rate is on the order of the window equilibration length (the number of packets it takes the window to regain its original size after a loss). If the pre-loss size is \( w \), equilibration takes roughly \( \frac{w}{2} \) packets so, for the Arpanet, the loss sensitivity

\[ \text{Tahoe RTT Estimation} \]
Slow start every time?!

• Losses have large effect on throughput
• Fast Recovery (TCP Reno)
  – Same as TCP Tahoe on Timeout: $w = 1$, slow start
  – On triple duplicate ACKs: $w = w/2$
  – Retransmit missing segment (fast retransmit)
  – Stay in Congestion Avoidance mode
Fast Recovery and Fast Retransmit

- Slow Start
- Fast retransmit
- AI/MD
3 Challenges Revisited

• Determining the available capacity in the first place
  – Exponential increase in congestion window
• Adjusting to changes in the available capacity
  – Slow probing, AIMD
• Sharing capacity between flows
  – AIMD
• Detecting Congestion
  – Timeout based on RTT
  – Triple duplicate acknowledgments
• Fast retransmit/Fast recovery
  – Reduces slow starts, timeouts