CSCI-1680
Transport Layer III
Congestion Control Strikes Back

Rodrigo Fonseca

Based partly on lecture notes by David Mazières, Phil Levis, John Jannotti, Ion Stoica
Last Time

- Flow Control
- Congestion Control
Today

• More TCP Fun!
• Congestion Control Continued
  – Quick Review
  – RTT Estimation
• TCP Friendliness
  – Equation Based Rate Control
• TCP on Lossy Links
• Congestion Control versus Avoidance
  – Getting help from the network
• Cheating TCP
Quick Review

• **Flow Control:**
  – Receiver sets Advertised Window

• **Congestion Control**
  – Two states: Slow Start (SS) and Congestion Avoidance (CA)
  – A window size threshold governs the state transition
    • Window <= ssthresh: SS
    • Window > ssthresh: Congestion Avoidance
  – States differ in how they respond to ACKs
    • Slow start: +1 w per RTT (Exponential increase)
    • Congestion Avoidance: +1 MSS per RTT (Additive increase)
  – On loss event: set ssthresh = w/2, w = 1, slow start
AIMD

Fair: \( A = B \)

Efficient: \( A + B = C \)
States differ in how they respond to acks

- **Slow start: double w in one RTT**
  - There are w/MSS segments (and acks) per RTT
  - Increase w per RTT \(\rightarrow\) how much to increase per ack?
    - \(w / (w/MSS) = MSS\)

- **AIMD: Add 1 MSS per RTT**
  - \(MSS/(w/MSS) = MSS^2/w\) per received ACK
Putting it all together

- **Slow Start**
- **Timeout**
- **AIMD**
- **ssthresh**
- **AIMD**
Fast Recovery and Fast Retransmit

- Slow Start
- Fast retransmit
- AI/MD

Diagram showing the relationship between cwnd and time.
RTT Estimation

• 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
- $\text{DevRTT} = (1 - \beta) \times \text{DevRTT} + \beta | \text{SampleRTT} - \text{EstRTT} |$
  - 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 \( T \) accounts for RTT variation (see [5], section 5). The suggested \( T = 2 \) 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 \( T \) 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
Tahoe RTT Estimation

Figure 6: Performance of a Mean+Variance retransmit timer

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 ($\ll 1\%$) so it is probable that a packet loss is due to congestion in the network.

Unfortunately, with an infinite user population even exponential backoff won't guarantee stability (although it 'almost' does—see [1]). Fortunately, we don't (yet) have to deal with an infinite user population.

The phrase congestion collapse (describing a positive feedback instability due to poor retransmit timers) is again the coinage of John Nagle, this time from [23].

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 $w$, the time to take roughly $w^2/3$ packets so, for the Arpanet, the loss sensitivity
TCP Friendliness

• Can other protocols co-exist with TCP?
  – E.g., if you want to write a video streaming app using UDP, how to do congestion control?

1 UDP Flow at 10MBps
31 TCP Flows
Sharing a 10MBps link
TCP Friendliness

• Can other protocols co-exist with TCP?
  – E.g., if you want to write a video streaming app using UDP, how to do congestion control?

• Equation-based Congestion Control
  – Instead of implementing TCP’s CC, estimate the rate at which TCP would send. Function of what?
  – RTT, MSS, Loss

• Measure RTT, Loss, send at that rate!
TCP Throughput

• Assume a TCP congestion of window $W$ (segments), round-trip time of $RTT$, segment size $MSS$
  – Sending Rate $S = \frac{W \times MSS}{RTT} \quad (1)$

• Drop: $W = W/2$
  – grows by $MSS$ for $W/2 \ RTTs$, until another drop at $W \approx W$

• Average window then $0.75xS$
  – From (1), $S = 0.75 \frac{W \times MSS}{RTT} \quad (2)$

• Loss rate is 1 in number of packets between losses:
  – Loss $= 1 / (1 + (W/2 + W/2+1 + W/2 + 2 + \ldots + W))$
  – $= 1 / (3/8 \ W^2) \quad (3)$
TCP Throughput (cont)

- Loss = $8/(3W^2) \Rightarrow W = \sqrt{\frac{8}{3 \cdot Loss}} \quad (4)$

- Substituting (4) in (2), $S = 0.75 \ W \ MSS / RTT$,

Throughput $\approx 1.22 \times \frac{MSS}{RTT \cdot \sqrt{Loss}}$

- Equation-based rate control can be TCP friendly and have better properties, e.g., small jitter, fast ramp-up...
What Happens When Link is Lossy?

- Throughput $\approx 1 / \sqrt{\text{Loss}}$
What can we do about it?

• Two types of losses: congestion and corruption
• One option: mask corruption losses from TCP
  – Retransmissions at the link layer
  – E.g. Snoop TCP: intercept duplicate acknowledgments, retransmit locally, filter them from the sender
• Another option:
  – Tell the sender about the cause for the drop
  – Requires modification to the TCP endpoints
Congestion Avoidance

- **TCP creates congestion to then back off**
  - Queues at bottleneck link are often full: increased delay
  - Sawtooth pattern: jitter

- **Alternative strategy**
  - Predict when congestion is about to happen
  - Reduce rate early

- **Two approaches**
  - Host centric: TCP Vegas (won’t cover)
  - Router-centric: RED, DECBit
TCP Vegas

- Idea: source watches for sign that router’s queue is building up (e.g., sending rate flattens)
TCP Vegas

- **Compare Actual Rate (A) with Expected Rate (E)**
  - If E-A > β, decrease cwnd linearly: A isn’t responding
  - If E-A < α, increase cwnd linearly: Room for A to grow
Vegas

• Shorter router queues
• Lower jitter
• Problem:
  – Doesn’t compete well with Reno. Why?
  – Reacts earlier, Reno is more aggressive, ends up with higher bandwidth…
Help from the network

- What if routers could *tell* TCP that congestion is happening?
  - Congestion causes queues to grow: rate mismatch
- TCP responds to drops
- Idea: Random Early Drop (RED)
  - Rather than wait for queue to become full, drop packet with some probability that increases with queue length
  - TCP will react by reducing cwnd
  - Could also mark instead of dropping: ECN
RED Details

• Compute average queue length (EWMA)
  – Don’t want to react to very quick fluctuations
RED Drop Probability

- Define two thresholds: MinThresh, MaxThresh
- Drop probability:

  \[
  P_{\text{Temp}} = \frac{\text{MaxP} \cdot \text{AvgLen} \cdot \text{MinThreshold}}{\text{MaxThreshold} \cdot \text{MinThreshold}}
  \]

  \[
  P = \frac{P_{\text{Temp}}}{1 + \text{count} \cdot P_{\text{Temp}}}
  \]

- Improvements to spread drops (see book)
RED Advantages

• Probability of dropping a packet of a particular flow is roughly proportional to the share of the bandwidth that flow is currently getting
• Higher network utilization with low delays
• Average queue length small, but can absorb bursts
• ECN
  – Similar to RED, but router sets bit in the packet
  – Must be supported by both ends
  – Avoids retransmissions optionally dropped packets
What happens if not everyone cooperates?

• TCP works extremely well when its assumptions are valid
  – All flows correctly implement congestion control
  – Losses are due to congestion
Cheating TCP

• Three possible ways to cheat
  – Increasing cwnd faster
  – Large initial cwnd
  – Opening many connections
  – Ack Division Attack
Increasing cwnd Faster

- $x$ increases by 2 per RTT
- $y$ increases by 1 per RTT

Figure from Walrand, Berkeley EECS 122, 2003
Larger Initial Window

Figure from Walrand, Berkeley EECS 122, 2003
Open Many Connections

- Web Browser: has to download $k$ objects for a page
  - Open many connections or download sequentially?

- Assume:
  - A opens 10 connections to B
  - B opens 1 connection to E

- TCP is fair among connections
  - A gets 10 times more bandwidth than B

Figure from Walrand, Berkeley EECS 122, 2003
Exploiting Implicit Assumptions

• Savage, et al., CCR 1999:
  – “TCP Congestion Control with a Misbehaving Receiver”

• Exploits ambiguity in meaning of ACK
  – ACKs can specify any byte range for error control
  – Congestion control assumes ACKs cover entire sent segments

• What if you send multiple ACKs per segment?
ACK Division Attack

- **Receiver:** "upon receiving a segment with $N$ bytes, divide the bytes in $M$ groups and acknowledge each group separately"
- **Sender will grow window $M$ times faster**
- **Could cause growth to 4GB in 4 RTTs!**
  - $M = N = 1460$
TCP Daytona!

3.1 ACK division
The TCP Daytona ACK division algorithm adds 24 lines of code that divide each new outgoing ACK into many ACKs for smaller extents of the sequence space. Half of the new code is dedicated to ensuring that the number of outgoing ACKs is no more than should be needed to coerce a sender in slow start to saturate our test machine's 100Mbps Ethernet interface.

Figure 4 shows client-side TCP sequence number plots of our test machine making an HTTP request for the index.html object from cnn.com, with our ACK division attack enabled. This figure spans the entire transaction, beginning with the TCP handshake that starts at 0ms and ends at around 70ms, when the HTTP request is sent. The first HTTP data from the server arrives at around 140ms.

This figure shows that, when this attack is enabled, the many small ACKs sent around 140ms convince the Web server to unleash the entire remainder of the document in a single burst; this data arrives exactly one round-trip time later. By contrast, with the normal TCP implementation, the server spreads out the data over the next four round-trip times. In general, as this figure suggests, this attack can convince a TCP sender to send all of its data in a single burst.

3.2 DupACK spoofing
The TCP Daytona DupACK spoofing attack is implemented by 11 lines of code that cause the receiver to send sufficient duplicate ACKs such that the sender (re-)enters fast recovery and fills the receiver's advertised flow control window each round-trip time.

Figure 5 shows another client-side plot of the same HTTP request, this time with the DupACK spoofing attack superimposed on a normal transfer. The many duplicate ACKs that the receiver sends at around 140ms cause the sender to enter fast recovery and transmit the rest of the data, which arrives at around 210ms. Were there more data, the flurry of duplicate ACKs sent at 210ms-230ms would elicit another burst from the sender. Since there is no more new data, the sender simply fills in the hole it perceives; this segment arrives at around 290ms. This figure illustrates how the DupACK spoofing attack can achieve performance essentially equivalent to the ACK division attack – namely, both attacks can convince the sender to empty its entire send buffer in a single burst.

3.3 Optimistic ACKing
The TCP Daytona implementation of optimistic ACKing consists of 45 lines of code. Because acknowledging data that has not arrived is a fundamentally tricky business, we chose a very simple implementation as a proof of concept. When a TCP connection for an HTTP or FTP client receives its first data, we set a timer to expire every 10ms. Any interval would do, but we chose 10ms because it is the smallest interval that Linux 2.2.10 supports on the Intel PC platform. Whenever this periodic timer expires, or a new data segment arrives, our receiver sends a new optimistic ACK for one MSS beyond the previous optimistic ACK.
Defense

• **Appropriate Byte Counting**
  – [RFC3465 (2003), RFC 5681 (2009)]
  – In slow start, cwnd += min (N, MSS)
where N is the number of newly acknowledged bytes in the received ACK
Cheating TCP and Game Theory

A
D

\( (x, y) \)

\( \text{Too aggressive} \)
\( \Rightarrow \text{Losses} \)
\( \Rightarrow \text{Throughput falls} \)

<table>
<thead>
<tr>
<th></th>
<th>( 22, 22 )</th>
<th>( 10, 35 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>Increases by 1</td>
<td>Increases by 5</td>
</tr>
<tr>
<td>( D )</td>
<td>Increases by 1</td>
<td>Increases by 5</td>
</tr>
<tr>
<td>( B )</td>
<td>( 35, 10 )</td>
<td>( 15, 15 )</td>
</tr>
</tbody>
</table>

Individual incentives: cheating pays
Social incentives: better off without cheating

Classic PD: resolution depends on accountability
An alternative for reliability

• **Erasure coding**
  – Assume you can detect errors
  – Code is designed to tolerate entire missing packets
    • Collisions, noise, drops because of bit errors
  – Forward error correction

• **Examples:** Reed-Solomon codes, LT Codes, Raptor Codes

• **Property:**
  – From K source frames, produce B > K encoded frames
  – Receiver can reconstruct source with *any* K’ frames, with K’ *slightly* larger than K
  – Some codes can make B as large as needed, on the fly
LT Codes

• **Luby Transform Codes**
  – Michael Luby, circa 1998

• **Encoder: repeat B times**
  1. Pick a degree $d$
  2. Randomly select $d$ source blocks. Encoded block $t_n = \text{XOR or selected blocks}$
LT Decoder

- Find an encoded block $t_n$ with $d=1$
- Set $s_n = t_n$
- For all other blocks $t'_n$ that include $s_n$,
  set $t'_n = t'_n \oplus s_n$
- Delete $s_n$ from all encoding lists
- Finish if
  1. You decode all source blocks, or
  2. You run out of blocks of degree 1
Next Time

- Move into the application layer
- DNS, Web, Security, and more…
Backup slides

• We didn’t cover these in lecture: won’t be in the exam, but you might be interested 😊
More help from the network

• **Problem: still vulnerable to malicious flows!**
  – RED will drop packets from large flows preferentially, but they don’t have to respond appropriately

• **Idea: Multiple Queues (one per flow)**
  – Serve queues in Round-Robin
  – Nagle (1987)
  – Good: protects against misbehaving flows
  – Disadvantage?
  – Flows with larger packets get higher bandwidth
Solution

• Bit-by-bit round robing
• Can we do this?
  – No, packets cannot be preempted!
• We can only approximate it…
• Define a fluid flow system as one where flows are served bit-by-bit
• Simulate \( ff \), and serve packets in the order in which they would finish in the \( ff \) system
• Each flow will receive exactly its fair share
Example

Flow 1 (arrival traffic)

Flow 2 (arrival traffic)

Service in fluid flow system

Packet system
Implementing FQ

• Suppose clock ticks with each bit transmitted
  – (RR, among all active flows)
• $P_i$ is the length of the packet
• $S_i$ is packet $i$’s start of transmission time
• $F_i$ is packet $i$’s end of transmission time
• $F_i = S_i + P_i$

• When does router start transmitting packet $i$?
  – If arrived before $F_{i-1}$, $S_i = F_{i-1}$
  – If no current packet for this flow, start when packet arrives (call this $A_i$): $S_i = A_i$

• Thus, $F_i = \max(F_{i-1}, A_i) + P_i$
Fair Queueing

• **Across all flows**
  – Calculate $F_i$ for each packet that arrives on each flow
  – Next packet to transmit is that with the lowest $F_i$
  – Clock rate depends on the number of flows

• **Advantages**
  – Achieves *max-min fairness*, independent of sources
  – Work conserving

• **Disadvantages**
  – Requires non-trivial support from routers
  – Requires reliable identification of flows
  – Not perfect: can’t preempt packets
Fair Queueing Example

- 10Mbps link, 1 10Mbps UDP, 31 TCPs
Big Picture

• **Fair Queuing doesn’t eliminate congestion: just manages it**

• **You need both, ideally:**
  – End-host congestion control to adapt
  – Router congestion control to provide isolation