CSCI-1680
Transport Layer II

Data over TCP: Congestion Control I

Nick DeMarinis

Based partly on lecture notes by Rodrigo Fonseca, David Mazières, Phil Levis, John Jannotti
Administrivia

• If you haven’t signed up for a TCP milestone meeting, do so!
  – Due by end meeting slots on Friday, April 15
  – If you can’t find a slot that works, let us know BEFORE the deadline
• Look for announcement on TCP gearup
• HW3: Out after milestone—prep for Milestone II
• After this class: done with transport layer
Topics for today

• TCP socket creation
• Small bits on congestion control
• Start of DNS?
TCP socket creation

• What happens when you run “a 80”?
• What happens when you connect to a socket?
The TCP checksum

- `checksum("pseudo header" + TCP header + data)`
- Same algorithm as IP checksum
Dealing with Congestion

Goals
• Determine initial network capacity
• Adjust sending rate as capacity changes

• How? Maintain two windows:
  – Advertised Window (from receiver)
  – Congestion window (cwnd)

\[
\text{Sending rate} = \min(\text{Advertised Window}, \ cwnd)
\]

• Ideally, want to have sending rate: \( \sim \frac{\text{Window}}{\text{RTT}} \)
Classical Congestion Control

• Loss-based: assume packet loss => congestion

• TCP Tahoe (1988)
  – Slow start, congestion avoidance, fast retransmit

• TCP Reno (1990)
  – TCP Tahoe + Fast recovery

• Many variations developed from this… (see optional readings)
Modes of operation

• Slow start (SS)
  – Determine initial window, recover after loss
• Congestion avoidance (CA)
  – Steady state, slowly probe for changes in capacity
After finishing a window, recompute cwnd:

- If no losses, cwnd = cwnd + MSS  
  - (Often written as cwnd += 1)
- If packets were lost: cwnd = cwnd/2

This is known as additive increase, multiplicative decrease (AIMD)

- Slowly increase capacity
- Dramatically scale back on loss
TCP sawtooth is specific to TCP Reno and related TCP implementations that share Reno's additive-increase/multiplicative-decrease mechanism.

During periods of no loss, TCP's cwnd increases linearly; when a loss occurs, TCP sets cwnd = cwnd/2.

This diagram is an idealization as when a loss occurs it takes the sender some time to discover it, perhaps as much as the TimeOut interval.

The fluctuation shown here in the red ceiling curve is somewhat arbitrary. If there are only one or two other competing senders, the ceiling variation may be quite dramatic, but with many concurrent senders the variations may be smoothed out.

For some TCP sawtooth graphs created through actual simulation, see 31.2.1 Graph of cwnd v time and 31.4.1 Some TCP Reno cwnd graphs.

19.1.1.1 A first look at fairness

The transit capacity of the path is more-or-less unvarying, as is the physical capacity of the queue at the bottleneck router. However, these capacities are also shared with other connections, which may come and go with time. This is why the ceiling does vary in real terms. If two other connections share a path with total capacity 60 packets, the "fairest" allocation might be for each connection to get about 20 packets as its share. If one of those other connections terminates, the two remaining ones might each rise to 30 packets. And if instead a fourth connection joins the mix, then after equilibrium is reached each connection might hope for a fair share of 15 packets.

Will this kind of "fair" allocation actually happen? Or might we end up with one connection getting 90% of the bandwidth while two others each get 5%?

Chiu and Jain [CJ89] showed that the additive-increase/multiplicative-decrease algorithm does indeed converge to roughly equal bandwidth sharing when two connections have a common bottleneck link, provided also that...
Slow Start

After finishing a window
• \( cwnd = cwnd \times 2 \)
• Continue doing this until you experience a loss

• After first loss, keep slow-start threshold (ssthresh):
  – If window < ssthresh: slow-start
  – If window > ssthresh: congestion avoidance
• After first loss: ssthresh = \( \frac{cwnd}{2} \)
19.2.3 Slow-Start Multiple Drop Example

Slow start has the potential to cause multiple dropped packets at the bottleneck link; packet losses continue for quite some time because the TCP sender is slow to discover them. The network topology is as follows, where the A–R link is infinitely fast and the R–B link has a bandwidth in the R → B direction of 1 packet/ms.

Assume that R has a queue capacity of 100, not including the packet it is currently forwarding to B, and that ACKs travel instantly from B back to A. In this and later examples we will continue to use the Data[N]/ACK[N] terminology of 8.2 Sliding Windows, beginning with N=1; TCP numbering is not done quite this way but the distinction is inconsequential.

When A uses slow-start here, the successive windowfuls will almost immediately begin to overlap. A will send one packet at T=0; it will be delivered at T=1. The ACK will travel instantly to A, at which point A will send two packets. From this point on, ACKs will arrive regularly at A at a rate of one per second. Here is a brief chart:

TCP Tahoe Sawtooth, red curve represents the network capacity
Slow Start is used after each packet loss until ssthresh is reached
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

• Which one is worse?
Putting it all together

- **cwnd**: Over time, the congestion window (cwnd) increases from slow start to AIMD (Additive Increase, Multiplicative Decrease) and back to AIMD again, with timeouts and ssthresh (slow start threshold) points indicated.
- **Timeout**: Points where the timeout occurs, resetting the process.
- **AIMD**: The AIMD phase is shown with arrows indicating the increase and decrease of the cwnd.

The graph illustrates the dynamic behavior of the congestion control protocol over time.
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
- Why 3 dup-acks instead of just 1?
This is just the beginning...

Lots of congestion control schemes, with different strategies/goals:

• Tahoe (1988)
• Reno (1990)
• Vegas (1994): Detect based on RTT
• New Reno: Better recovery multiple losses
• Cubic (2006): Linux default, window size scales by cubic function
• BBR (2016): Used by Google, measures bandwidth/RTT
BBR

From: https://labs.ripe.net/Members/gih/bbr-tcp
TCP State Diagram

- **CLOSED** (Start)
  - LISTEN/-
  - CONNECT/SYN (Step 1 of the 3-way-handshake)

- **LISTEN**
  - SYN/SYN+ACK
  - LISTEN/-

- **SYN SENT**
  - SEND/SYN
  - SYN+ACK/ACK (simultaneous open)

- **SYN RECEIVED**
  - SYN/SYN+ACK
  - RST/-
  - SYN/SYN+ACK (Step 2 of the 3-way-handshake)

- **ESTABLISHED**
  - SYN+ACK/ACK
  - Data exchange occurs

- **FIN WAIT 1**
  - FIN/ACK
  - FIN/ACK

- **CLOSING**
  - FIN/ACK
  - FIN+ACK/ACK

- **TIME WAIT**
  - ACK/-
  - Timeout

- **FIN WAIT 2**
  - FIN/ACK

- **CLOSE WAIT**
  - LAST ACK

**Active CLOSE**
- FIN/ACK

**Passive CLOSE**
- FIN/ACK

- **CLOSED**
  - FIN/ACK

Unusual event client/receiver path

Unusual event server/sender path

**TIME WAIT**
- Active CLOSE
- Passive CLOSE

**LAST ACK**
- FIN/ACK
TCP Header

```
+-------------+-------------+-------------+-------------+
<p>| | | | |
|             |             |             |             |
| 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 |
|-------------|-------------|-------------|-------------|
| Offset      | Reserved    | R | C | S | Y | I | Window |
|-------------|-------------|-------------|-------------|
| Data        | U | A | P | R | S | F |        |
|-------------|-------------|-------------|-------------|
| Checksum    | G | K | H | T | N | N |        |
|-------------|-------------|-------------|-------------|
| Acknowledgment Number |         |
|-------------|-------------|-------------|-------------|</p>
<table>
<thead>
<tr>
<th>Source Port</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgment Number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Destination Port</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Sequence Number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Options</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Padding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Padding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Padding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Padding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
```

0                   1                   2                   3
Extra congestion control content
Putting it all together

- cwnd
- Time
- ssthresh
- AIMD

Slow Start → Timeout
AIMD
Slow Start → Timeout
AIMD
Fast Recovery and Fast Retransmit

- Slow Start
- Fast retransmit
- AI/MD
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 \( S = \frac{W \times MSS}{RTT} \) (1)

• Drop: \( W = \frac{W}{2} \)
  – grows by MSS for \( \frac{W}{2} \) RTTs, until another drop at \( W \approx W \)

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

• Loss rate is 1 in number of packets between losses:
  – Loss = \( \frac{1}{1 + \left( \frac{W}{2} + \frac{W}{2} + 1 + \frac{W}{2} + 2 + \ldots + W \right)} \)
     = \( \frac{1}{\left( \frac{3}{8} W^2 \right)} \) (3)
TCP Throughput (cont)

- Loss = $\frac{8}{3W^2}$ [equation (4)]

$$\Rightarrow W = \sqrt{\frac{8}{3 \cdot \text{Loss}}}$$

- Substituting (4) in (2), $S = 0.75 \times \frac{W \cdot \text{MSS}}{\text{RTT}}$

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

- 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

• Other approaches
  – Delay Based: TCP Vegas (not covered)
  – Better model of congestion: BBR
  – Router-centric: RED, ECN, DECBit, DCTCP
Another view of Congestion Control

Tput = InFlight / RTT_{prop}

Another view of Congestion Control

Round Trip Time

Bytes in Flight

Throughput

Bytes in Flight

RTT$_{prop}$

BDP

Bottleneck BW

Slope = 1/RTT$_{prop}$
Another view of Congestion Control

Throughput

Bytes in Flight

Round Trip Time

RTT_{prop}

Slope = 1/BW

Bytes in Flight

Slope = 1/RTT_{prop}

BDP

Bottleneck BW

BDP+Bottleneck Queue

RTT_{prop}
Another view of Congestion Control

Throughput

Bytes in Flight

Round Trip Time

Slope = 1/BW

Loss-based CC

Ideal Operating Point

Slope = 1/\text{RTT}_\text{prop}

RTT

Bottleneck BW

Ideal Operating Point

Slope = 1/\text{RTT}_\text{prop}

BDP

Bytes in Flight

BDP + Bottleneck Queue
BBR

• Problem: can’t measure both $\text{RTT}_\text{prop}$ and Bottleneck BW at the same time
• BBR:
  – Slow start
  – Measure throughput when RTT starts to increase
  – Measure RTT when throughput is still increasing
  – Pace packets at the BDP
  – Probe by sending faster for 1RTT, then slower to compensate
BBR

From: https://labs.ripe.net/Members/gh/bbr-tcp
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
• Compute average queue length (EWMA)
  – Don’t want to react to very quick fluctuations
RED Drop Probability

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

\[
P_{\text{drop}} = \frac{\text{TempP}}{(1 \cdot \text{count} \cdot \text{TempP})}
\]

- Drop Probability Curve:
  \[
  P_{\text{drop}} = \frac{\text{MaxP}}{\text{MinThresh}}
  \]
  \[
  P_{\text{drop}} = 1.0
  \]

- 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

• Possible ways to cheat
  – Increasing cwnd faster
  – Large initial cwnd
  – Opening many connections
  – Ack Division Attack
Larger Initial Window

x starts SS with cwnd = 4
y starts SS with cwnd = 1

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!

Figure 4: The TCP Daytona ACK division attack convinces the TCP sender to send all but the first few segments of a document in a single burst.

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, without 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. Figure 5 shows that, when this attack is enabled, the many duplicate ACKs sent 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 += \text{min} (N, MSS)
  
where N is the number of newly acknowledged bytes in the received ACK
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…
Fair Queueing

- 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
• 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