TCP Performance

EE 122: Intro to Communication Networks
Fall 2006 (MW 4-5:30 in Donner 155)

Vern Paxson
TAs: Dilip Antony Joseph and Sukun Kim

http://inst.eecs.berkeley.edu/~ee122/

Materials with thanks to Jennifer Rexford, Ion Stoica, and colleagues at Princeton and UC Berkeley

Announcements

- Homework #4 should be out tonight

- Next lecture: network performance
Goals of Today’s Lecture

• A look at a range of TCP mechanisms that influence its behavior & performance:
  – Window limitations
  – Timeout, slow start, exponential backoff
  – Acking policies
  – Fast Retransmit
  – Fast Recovery (full AIMD)
  – Window scaling

• Visualized using time-sequence plots

Time-Sequence Plots

• Powerful tool for visualizing transport protocol dynamics

• Plot sequence number (Y axis) vs. time (X axis)
  – For data packet, sequence number = highest byte carried in packet, so header seqno + payload length
  – For ack, it’s just the ack seqno in the header

• Where do you get the data for the plot?
  – Record trace of connection using tcpdump

• Illuminates performance achieved & why
  – As well as problems
    • Both network and in endpoint stacks
    • (and also beware measurement errors)
Example of Time-Sequence Plot

Hollow squares = Acks
Solid squares = Data
MSS
RTT
Window

Slope gives overall throughput (bytes/sec)
Same Connection - Why So Different?

Note: Receiver acks every other segment

Delayed Acknowledgments

- Receiver generally delays sending an ACK
  - Upon receiving a packet, sets a timer
    - Typically, 200 msec; at most, 500 msec
  - If application generates data, go ahead and send
    - And piggyback the acknowledgment
  - If the timer expires, send a (non-piggybacked) ACK
  - If out-of-order segment arrives, immediately ack
  - (if available window changes, send an ACK)

- Limiting the wait
  - Receiver supposed to ACK at least every second full-sized packet ("ack every other")
    - This is the usual case for "streaming" transfers
Performance Effects of Acking Policies

• How do delayed ACKs affect performance?
  – Increases RTT
  – Window slides a bit later ⇒ throughput a bit lower

• How does ack-every-other affect performance?
  – If sender adjusts CWND on incoming ACKs, then CWND opens more slowly
    • In slow start, 50% increase/RTT rather than 100%
    • In congestion avoidance, +1 MSS / 2 RTT, not +1 MSS / RTT

• What does this suggest about how a receiver might cheat and speed up a transfer?

ACK-splitting

- Rule: grow window by one full-sized packet for each valid ACK received
- Send $M$ (distinct) ACKs for one packet
- Growth factor proportional to $M$
- What’s the fix?
10 line change to Linux TCP

Page fetch from CNN.com

Sequence Plot for an Entire Transfer

Why does this transfer only achieve 50 KB/s?
Beginning of Transfer - Slow Start

Q: What is this first small packet?
A: The initial SYN

Q: Why a gap here?
A: That packet is still in flight

Q: Why the long RTT?
A: Delayed ack (just for 1 MSS)

Mid-Transfer: Self-Clocking

Each flight of packets has the same shape!

As do the ACKs …
Sliding Window induces **Self-Clocking**

Bottleneck link stretches out data packets

Sender

Next flight of packets goes out with bottleneck’s spacing

Receiver

Spacing is preserved upon arrival... and hence in the ACKs

(From [JK88])

---

**Mid-Transfer: Why Doesn’t CWND Grow?**

---
Receiver Window = 8 MSS

Circles show advertised window

Same Transfer w/ Large Recv. Window

Throughput doubles …
But why isn’t sender using entire window?
**Sliding Window**

- Allow a larger amount of data “in flight”
  - Allow sender to get ahead of the receiver
  - … though not too far ahead

---

**Same Transfer w/ Large Recv. Window**

Sender’s window is 9.2 KB (about double receiver’s)
Same Transfer w/ Large Snd. Window

Short-term throughput again goes up ...

But what happens here?

5 Minute Break

Questions Before We Proceed?
Detecting Loss: Timeout

Detecting Loss: Timeout - Recv. View
Timeout and Effect of **SSThresh**

Prior to timeout, CWND = 8 MSS

We finally increment CWND in Cong. Avoid

After timeout, CWND = 1 MSS, SSThresh = 4 MSS

Slow Start until CWND > 4 MSS, then Cong. Avoidance

Illustration of Exponential Backoff

RTO progresses 1.8*, 3.0, 6.0, 12.0 sec

63.8 sec

47.9 sec

23.8 sec

68000

70000

71000

Sequence #
How Many Packets Were Lost?

Answer: Zero!
**Fast Retransmission**

After pending data ack’d, slow start. CWND = 2 MSS since ACK arrival incremented it by MSS.

Window stays at 5 MSS ⇒ transition to Congestion Avoidance.

Third dup triggers retransmission.

---

**Same Fast Retransmission @Recv.**

Again, arrivals much more smooth due to bottleneck shaping.

What happened here?
Effectiveness of Fast Retransmit

• When does Fast Retransmit work best?
  – Long data transfers
    • High likelihood of many packets in flight
  – High window size
    • High likelihood of many packets in flight
  – Low burstiness in packet losses
    • Higher likelihood that later packets arrive successfully

• Implications for Web traffic
  – Most Web transfers are short (e.g., 10 packets)
    • Short HTML files or small images
  – So, often there aren’t many packets in flight
  – … making fast retransmit less likely to “kick in”
  – Forcing users to like “reload” more often…

Fast Retransmit & Loss of Performance

Big juicy pre-loss throughput takes a large hit, even given slow-start to open up CWND again
**Fast Recovery Algorithm**

As more dups arrive, CWND is “inflated”, eventually allowing more data to be sent.

After 3 dups, subsequent dups indicate packets are leaving the network.

After pending data ack’d, window is set to SSTHRESH (deflated).

Connection proceeds at half prev. rate.

---

**Fast Rexmit/Recovery with > 1 Loss**

Packet resent due to Fast Retransmit is ack’d, but not beyond it. At this point, progress stalls ...

.... until timeout (followed by Slow Start).
Same From Receiver Perspective

Many packets are retransmitted unnecessarily (which also causes more dup's)

Significant holes in original arrivals

Fix: Selective Acknowledgment (SACK) option. Sender learns just what receiver has received. Avoids both timeout on second loss, and unnecessary retransmits.

Summary of TCP Mechanisms

• Delayed Acknowledgment
  – Lessens overhead (40 bytes per ACK)
  – But can cause CWND to grow more slowly

• Fast Retransmit
  – NACK-based loss detection in 1 RTT
  – Avoids timeout delay
  – AIMD after subsequent Slow Start reaches SSTHRESH

• Fast Recovery
  – Avoids needing to Slow Start after Fast Retransmit
  – True AIMD

• SACK
  – Both speeds recovery and avoids unnecessary retransmit.
A Precautionary Tale

What’s strange about this ACK?

Different Problem. What’s up now?

Is this a “measurement drop”? What’s the evidence?
Going Fast

- Q: on a path with RTT = 100 msec, what’s the absolute fastest rate that TCP can achieve?

- Q: what’s the absolute largest sliding window that TCP can use?
  - A: advertised window is 16 bits ⇒ 65,535 bytes
  - Thus: max speed = 65,535 bytes / 100 msec = 655 KB/s

- Q: how can we fix this problem?
  - A: we need a larger window

- Q: how do we make the window larger?
  - A: using a TCP option

Window Scaling Option (RFC 1323)

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| HdrLen specifies 4 bytes of options. |
| Kind=3 indicates “Window Scaling”. |
| Kind=0 indicates “end of options”. |

<table>
<thead>
<tr>
<th>Len=6</th>
<th>Flags</th>
<th>Advertised window</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Checksum</th>
<th>Urgent pointer</th>
</tr>
</thead>
</table>

Data
Window Scaling, con’t

• Sent in initial SYN

• If server’s SYN-ACK also includes a Window Scaling option, then scaling is in effect
  – The server including the option confirms its use

• shift.cnt specifies scaling factor for units used in window advertisement
  – E.g., shift.cnt = 5 ⇒ advertised window is $2^5 = 32$-byte units

Window Scaling, con’t

• Q: Now how large can the window be?

• A: Clearly, must not exceed $2^{32}$ …
  – If it does, then can’t disambiguate data in flight
  – So, scaling ≤ 16

• In fact, somewhat subtle requirements limit window to $2^{30}$ to allow receiver to determine whether data fits in the offered window
  – So, scaling ≤ 14
Window Scaling, con’t

• Now we can go fast. Suppose in a high-speed LAN with RTT = 1 msec we can transmit at 1 GB/s.

• What problem arises if packets are occasionally delayed in the network for 10 msec?

• Sequence number wrap: can’t tell earlier, delayed segments from later instances.

• Fix: another TCP option to associate (high-res) timestamps with TCP segments
  – Essentially, adds more bits to sequence space
  – (Side effect: no more need for Karn/Partridge restriction not to compute RTT for ACKs of retransmitted packets)

Summary

• Time-Sequence plots provide a powerful tool for visualizing TCP behavior & performance

• Spectrum of TCP mechanisms influence performance
  – Advertised window, sender window
  – Timeout, slow start, exponential backoff
  – Acking policy (delayed; ack-splitting)
  – Fast Retransmit (avoid RTO stall)
  – Fast Recovery (full AIMD)
  – Window scaling (required for large bandwidth-delay product)

• Next lecture: a broader view of performance