TCP Protocol Details, Part 2

Vermont Technical College
Peter C. Chapin

Receiver Window

- The receiver window manages flow control.
  - Receiver adjusts size to reflect buffer space.
  - Sends window size updates via “Window” field in TCP segment header.
- Sender won't send more data than receiver can handle.
  - ... even in the case where receiving application is busy elsewhere.

Sources of Slowness

- Receiver
  - Slow computer
  - Distracted program
    - Dealing with other tasks...
    - Processing received data is complicated...
  - *Receiver buffer fills and receiver window shrinks.*
- Network
  - Slow links
  - High traffic
  - *How is this handled?*

Congestion Window

- TCP maintains a second window.
  - *Estimate of the network's capacity to transmit data*.
  - Sender must compute the size of this window
    - Based on implicit feedback from the receiver
      - Successful ACKs
      - Timeouts
    - *Assumption: Lost segments are due to network congestion* (is this really true?)
- Actual window used for transmission is the smallest of (receiver, congestion).

Slow Start

- Congestion window size (*cwnd*) starts small and grows to “probe” the network capacity.
  - In what follows “one segment” means the MSS used by the sender (typically 1460 bytes on ethernet).
- Initialize with *cwnd* = 1 segment.
- Increment *cwnd* by 1 segment for each segment acknowledged.
  - This increases *cwnd* exponentially!

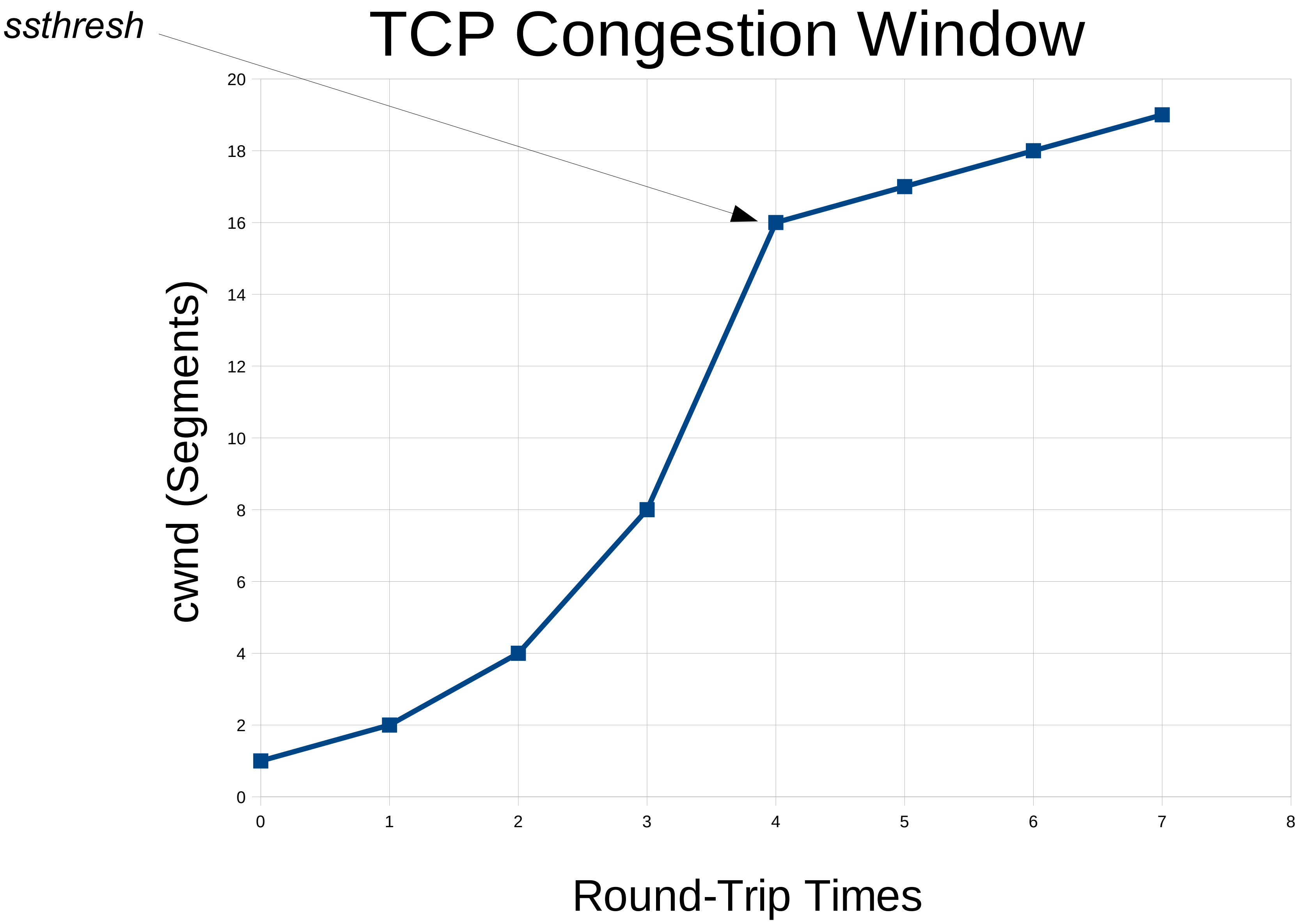
Exponentially?

- Consider...
  - Set *cwnd* = 1 segment. Send it.
  - Wait for ACK. Set *cwnd* = 2 segments. Send them.
  - After both ACKs...
    - Set *cwnd* = 2 + 1 + 1 = 4 segments. Send them.
  - After all four ACKS...
    - Set *cwnd* = 4 + 1 + 1 + 1 + 1 = 8 segments. Send them.
- In real life it is more complicated.
  - ACKs don't really arrive all together (in general).
  - TCP follows the same basic rule, however.

Slow Start Threshold

- A second value, *ssthresh*, defines when slow start ends and "congestion avoidance" begins.
  - After *cwnd* reaches *ssthresh*...
    - Increment *cwnd* by *1/cwnd* (as measured in segments) for each ACK.
      - Example: If *cwnd* = 4 segments, then add ¼ segment to *cwnd* in response to the next ACK.
      - Thus 4 ACKs needed to increase *cwnd* by 1 segment.
    - Thus *cwnd* increases by 1 for each round trip time regardless of segment count.
  - <u>Causes a linear increase of *cwnd*.</u>

Summary



ssthresh
TCP Congestion Window
cwnd (Segments)
Round-Trip Times
0
2
4
6
8
10
12
14
16
18
20
0
1
2
3
4
5
6
7
8

Timeout!

- When a timeout occurs...
  - *ssthresh* is set to ½ the current *cwnd* value.
  - *cwnd* is set to 2.
  - Slow start begins again.
- TCP assumes timeout means data loss.
  - Backs off by reducing the congestion window size.
  - Begins probing the network again in case source of congestion is gone.

Remember...

- TCP uses the smallest of (receiver, congestion) windows.
  - Once *cwnd* exceeds the receiver window, flow is limited by receiver window size.
  - This is the normal case on a clear network.
- On a WAN, however, *cwnd* is often limiting.
- <u>Many details left out of this dicussion.</u>
  - See references slide at end of this slide group.

How Long to Timeout?

- Too long...
  - If TCP waits too long to retransmit a lost segment time is wasted.
  - Slows down transmission.
- Too short...
  - If TCP doesn't wait long enough, it may retransmit unnecessarily.
  - Clogs the network.
  - Wastes bandwidth.

Round Trip Time?

- How long is a normal round trip?
  - LAN...
    - Transit time is sub-millisecond.
    - Usually steady.
  - WAN...
    - Transit time is multiple millisecond.
    - Often tens, hundreds, even thousands of milliseconds.
    - Often highly variable.
  - Computation time is usually short.
    - TCP acknowledges, application not involved.

RTT Estimation (Old)

- RFC-793 contains an algorithm for estimating round trip time (`RTT`).
  - Associate a timer with each outgoing segment.
  - When an ACK comes in, note the measured RTT for that segment (`M`).
  - Compute: $R_{new} = \alpha R_{old} + (1 - \alpha) M$
    - Where *α* is a scale factor (typically 0.9). *R* is an estimate of the RTT.
  - Compute timeout: $T = R_{new} \beta$
    - Where *β* is another scale factor (typically 2).

Problems

- The previous algorithm is not that great.
  - Can't keep up with changes.
  - Doesn't deal with highly variable RTT values.
  - Tends to cause many unnecessary retransmissions.
- What is needed is a way to account for the degree of variability in the RTT.

Jacobson's Algorithm

- Compute both RTT and “deviation” estimates.
  - Compute $E_r = M - R_{old}$
    - Note that the error value is signed.
  - Compute $R_{new} = R_{old} + g\,E_r$
    - Here *g* is typically 1/8.
  - Compute $D_{new} = D_{old} + h\left(|E_r| - D_{old}\right)$
    - Here *h* is typically ¼. *D* is an estimate of the deviation in observed RTT values.
  - Compute $T_{new} = R_{new} + 4\,D_{new}$
    - Time is RTT with extra to account for variability of RTT.
- Note that computations above are easy.

How is RTT Measured?

- Both methods described so far depend on `M`, the measured RTT. Where does that come from?
  - For each segment sent…
    - Note sequence # just off the end of the segment. Note time with a high resolution clock.
  - When an ACK covering that sequence number first arrives…
    - Note time on high resolution clock, subtract previously recorded time.

TSopt

- Managing timestamp data is a burden
  - Many segments in flight; each has a different send timestamp. When an ACK arrives, must figure out to which segment(s) it applies, etc.
  - RFC-7323 discusses the TSopt option
    - Sending timestamp installed in outgoing segment (TSval)
    - ACKs echo this value (TSecr)
    - TCP need not maintain a database of send timestamps for all in-flight segments.
    - No need to synchronize clocks! Echoed timestamps are in terms of the sender’s clock.

TCP Performance

	DATA Octets	ACK Octets
Preamble	8	8
Ethernet Header	14	14
IP Header	20	20
TCP Header	20	20
Data	1460	0
Pad	0	6
FCS (CRC)	4	4
Interframe Gap	12	12
TOTAL	**1538**	**84**

Needed to meet ethernet minimum of 64 octets per frame

9.6 microseconds on 10 Mbps ethernet.

Performance Computation

- Assume one ACK for every two data segments
  - In real life there are many possibilities.
- Assume 10 Mbps ethernet.

Real data

Raw data rate (1,250,000 octets/s)

Two data segments per ACK

$$\frac{2(1460)}{2(1538)+84} * \frac{10,000,000}{8} = 1,155,063\, octets/s$$

Data+overhead

ACK overhead

Actual data rate

Interactive TCP

- So far we have assumed we are transferring a large file… a steady stream of data primarily in one direction.
- Interactive sessions are different
  - One byte at a time (each character typed)
  - Small bursts of data bidirectionally
  - Think: terminal session such as SSH or telnet.

Small Packet Problem

- It goes like this:
  - User types character
  - TCP sends segment with one byte of data
  - *Huge overhead!*
- In absolute terms such segments are small…
  - … but if there are many of them they can create excessive congestion (particularly on slow links)

Nagle’s Algorithm

- Batch small writes to the connection. Send them all at once (several keystrokes in one segment).
- *Do not send data if there is previously unACKed data in flight. Instead buffer it.*
    - Unless… there is MSS data waiting in the buffer.
    - The last point allows smooth flow in the case of a file transfer.
- Degrades to stop-and-wait when interactive
    - Not a problem: RTT small by human standards

RTT Small?

- What about satellite links?
  - Geostationary satellites are 22,200 miles above Earth’s surface…
  - At the speed of light it takes ~240 ms to go up and back…
  - … plus the time on the terrestrial Internet.
  - … yields RTT on the order of ½ sec+

With Non-Local Echo

- Without Nagle's Algorithm…
  - Each letter typed appears about ½ second after you type it. When you stop typing, characters continue to appear as the echo catches up.
- With Nagle's Algorithm…
  - The letters appear about ½ second later, as before, but now in batches. When you stop typing, the last batch appears in about ½ second.
  - … BUT, only a fraction of the packets are sent.

References

- RFC-793: Transmission Control Protocol
- RFC-896: Congestion Control in IP/TCP Internetworks. (Describes Nagle's Algorithm for interactive connections)
- RFC-2581: TCP Congestion Control
- RFC-7323: TCP Extensions for High Performance
- http://en.wikipedia.org/wiki/Transmission_Control_Protocol