Congestion Avoidance and Control
Van Jacobson,
“Congestion Avoidance and Control”,
SIGCOMM 1988
Fixes to TCP in BSD
Handwaving arguments
Less rigorous math
Lots of “magical” hacks
We assume
- the sender always has data to send
- each packet is of the same size
- TCP is message-oriented
1986
TCP throughput from LBL to UC Berkeley (two hops) dropped from 32K bps to 40 bps.
Sending window = receiving window

No congestion control

Retransmit only when timeout
Congestion Collapse:
sender sends too fast
routers delay/drop packets
sender retransmit
no useful data getting through
Observation: a TCP connection should obey Conservation of Packets
In equilibrium state, a new packet is not inserted until an old packet leaves.
How could this principle be violated?
1. Never reaches equilibrium
2. Inject a packet before the next packet leaves
1. Getting to the equilibrium state
Equilibrium state: self-clocking
Figure 1: Window Flow Control ‘Self-clocking’
How to start the ‘clock’?
Slow Start
Add a new variable \textit{cwnd}.

Start/Restart: \textit{cwnd} = 1.

Upon receiving ACK, \textit{cwnd}++.

Send at most \textit{min(cwnd,rwin)}
Never send more than 2x the max possible rate.

(previously 200x is possible!)
Figure 4: Startup behavior of TCP with Slow-start
2. Inject a packet before the next packet leaves
2. Conservation at Equilibrium
Something’s wrong with TCP timer
Figure 5: Performance of an RFC793 retransmit timer
TCP (RFC793)

\[
R_i \leftarrow (1 - \alpha)R_{i-1} + (\alpha)M_i
\]

\[
RTO_i \leftarrow \beta R_i
\]

- \(R_i\) : smoothed RTT
- \(M_i\) : measured RTT
- \(RTO\) : timeout value
Variation in RTT when network is loaded
$\beta = 2$ (recommended) tolerates only 30% load
Idea: estimate the variation and use in calculating RTO
Measuring Variation

variance: costly (need to square)

mean error: simpler
\[ R_i \leftarrow (1 - \alpha)R_{i-1} + (\alpha)M_i \]
\[ R_i \leftarrow R_{i-1} + \alpha(M_i - R_{i-1}) \]
\[ V_i \leftarrow V_{i-1} + \alpha(|M_i - R_{i-1}| - V_{i-1}) \]
\[ RTO_i \leftarrow R_i + kV_i \]
To prevent spurious timeout,

\[ RTO_i > R_{i+1} \]
To pick a value of k, consider bandwidth-dominated link.
The diagram illustrates the concept of One Round Trip Time and One Packet Time. The numbers 1 through 15 represent different packets, with each packet categorized into different round trip times (0R, 1R, 2R, 3R). The packets are arranged in a series, indicating the sequence and time of transmission and reception.
R doubles each round during slow-start.
\[ RTO_i > R_{i+1} \]
\[ R_i + kV_i > 2R_i \]
\[ R_i + k(R_i - R_{i-1}) > 2R_i \]
\[ R_i + k(R_i - \frac{1}{2}R_i) > 2R_i \]
\[ k\left(\frac{1}{2}\right) > 1 \]
\[ k > 2 \]
\[ RTO_i = R_i + 4V_i \]
Figure 6: Performance of a Mean+Variance retransmit timer
Figure 5: Performance of an RFC793 retransmit timer
3. Moving towards new equilibrium when path changes
Idea: adjust cwnd when congestion happens
Assume: congestion leads to packet loss, leads to timeout.
On timeout, \(cwnd \not= 2\)

On ACK, \(cwnd += 1/cwnd\)
Why drop by half?

1. Slow-start:
   we know R/2 works

2. Steady state:
   a new flow probably?
Combining slow-start and congestion avoidance
TCP Tahoe
cwnd: “pipe size” probed

ssthresh: “pipe size” during equilibrium
new ack:
if (cwnd < ssthresh)
    cwnd += 1
else
    cwnd += 1/cwnd
timeout/3rd dup ack:
retransmit all unacked
$ssthresh = \frac{cwnd}{2}$
cwnd = 1
Improving TCP Tahoe:

Packets still getting through in dup ack -- no need to reset the clock!
TCP Reno
timeout:
retransmit all unacked
$ssthresh = cwnd/2$
cwnd = 1
3rd duplicate ACK: fast retransmission

(ie, retransmit 1st unack)

fast recovery

(details in Week 4)

ssthresh = cwnd = cwnd/2
AIMD
additive increase
multiplicative decrease