eecs 489 COMPUTER NETWORKS

Lectures 32:
More TCP Congestion Control

Issues

How does TCP congestion control performs over asymmetric links?

High speed and high bandwidth-delay product?
- to reach 10 Gbps requires packet loss rate of 1/90 minutes!

Short flows
- most flows are short
- most bytes are in long flows

How does TCP congestion control performs over wireless links?
- packet reordering fools fast retransmit
- loss not a good indicator of congestion

Effect of Asymmetry on TCP

Example of asymmetric networks:
- cable modem: 10 Mbps down, 512 Kbps up
- ADSL: 8 Mbps down, 1 Mbps up
- etc.

Fine for web surfing, but transfer can be slowed down by bandwidth asymmetry

Slower bandwidth on reverse path stretches out ACKs (ACK dilation), resulting in slower throughput on the forward path

Effect of Asymmetry on TCP

Maximum transmission speed determined by "normalized bandwidth ratio", the number of data packets per ACK, given forward and return bandwidth

$$\frac{\text{forward bandwidth/datasize}}{\text{reverse bandwidth/ACKsize}}$$

For example:
- 10 Mbps forward/100 Kbps reverse = bandwidth ratio of 100
- 1 KB data packet/40-byte ACK = data/ACK ratio of 25
- normalized bandwidth ratio = 100/25 = 4

⇒ if there are more than 1 ACK for every 4 packets, reverse bottleneck link will be saturated before forward link

⇒ use delayed cumulative ACK to reduce number of ACKs

TCP for High Speed Network

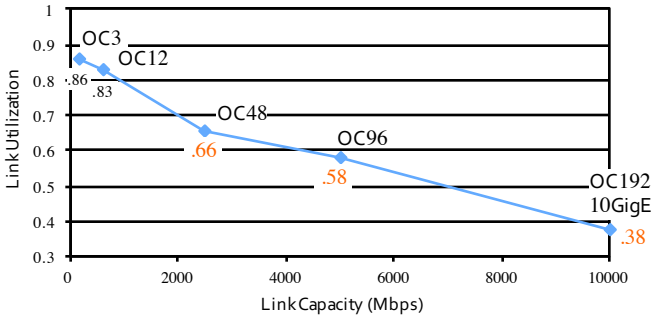
Multiple variants, we'll look at (CU)BIC

Let:
μ: 10 Gbps
RTT: 100 ms
MTU: 1250 bytes
bandwidth × delay product (BDP): ~100,000 packets

For TCP in linear increase to grow `cwnd` from 50,000 packets takes about 50,000 RTTs or 5,000 secs (1.4 hours)

[Rhee]

TCP Link Utilization Simulation



- μ: 155 Mbps (OC3), 622Mbps (OC12), 2.5Gbps (OC48), 5Gbps (OC96), 10Gbps (OC192, 10 GigE),
- 5 TCP connections, 100ms RTT, 1,000-byte packets

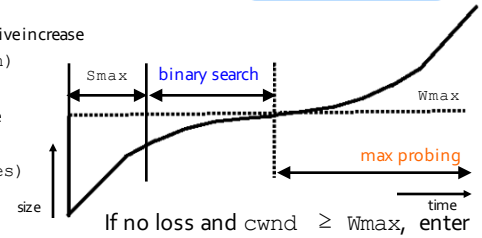
[Rhee]

BIC: Binary Increase Congestion Ctl

Upon loss:

```
Wmax = cwnd;
cwnd = Wmin = βWmax; // multiplicative decrease
while (Wmin <= Wmax){
 inc = (Wmin+Wmax)/2 - cwnd;
 if (inc > Smax)
   inc = Smax; // additive increase
 else if (inc < Smin)
   inc = Smin;
 // else logarithmic increase
 cwnd = cwnd + inc;
 if (no packet losses)
   Wmin = cwnd;
 else
   break;
} every RTT;
```

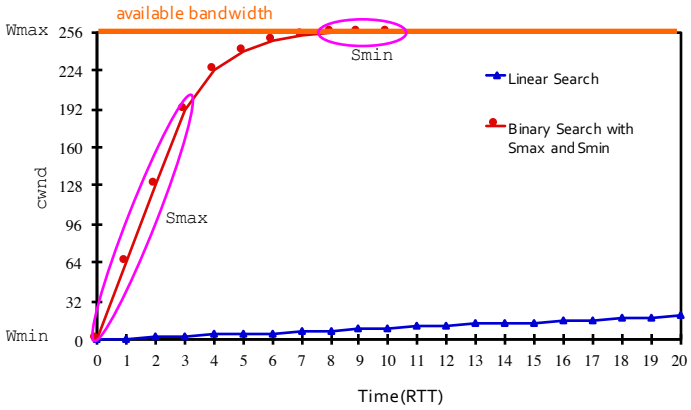
`Wmax`: max window
`Wmin`: min window
`Smax`: max increment
`Smin`: min increment



If no loss and `cwnd` ≥ `Wmax`, enter "max probing" phase: exponential increase, then additive increase

[Rhee]

BIC Performance



[Rhee]

BIC to CUBIC

BIC adopted as the default TCP congestion control module for Linux 2.6.8 (2004)

Undesirable characteristics of BIC:

- unfair to TCP under short-RTT or low-speed network
- different phases (linear increase, binary search, max probing) and use of window clamping parameters (`Smax`, `Smin`) complicated implementation and protocol analysis

CUBIC manages `cwnd` by a single growth function:

$$W(t) = C(t - K)^3 + W_{\max}$$

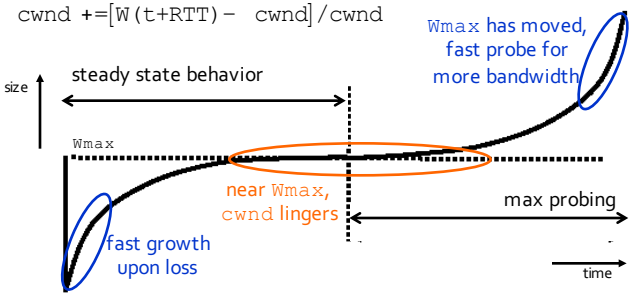
where C is a constant, t is time elapsed since last window reduction, and $K = \sqrt[3]{\frac{\beta W_{\max}}{C}}$, $0 < \beta < 1$

[Rhee]

CUBIC Function

$$W(t) = C(t - K)^3 + W_{\max}$$

`cwnd +=[W(t+RTT)- cwnd]/cwnd`



default Linux TCP congestion control since v. 2.6.18 (2006)
2.6.21 (2007): cubic root calculation reduced from 1,032 to 79 clocks
2.6.25-rc3 (2008): all window clamping variables removed

[Rhee]

Effectiveness of Fast Retransmit

When does Fast Retransmit work best?

- long data transfers
  - high likelihood of many packets in flight
- large window size
  - high likelihood of many packets in flight
- low burstiness in packet losses
  - higher likelihood that later packets cause dupACKs to be returned

Implications for Web traffic

- most Web transfers are short (e.g., 10 packets)
  - short HTML files or small images
- often there aren't that many packets in flight
  - making fast retransmit less likely to "kick in"
  - making you click "reload" more often... ☺

[Rexford]

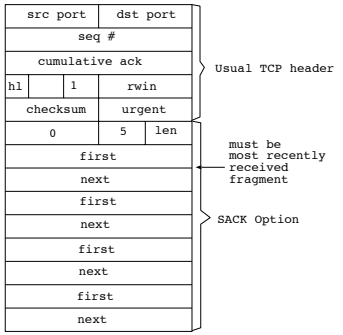
TCP SACK

Selective ACK (SACK) gives sender a better idea of which packets have been successfully delivered

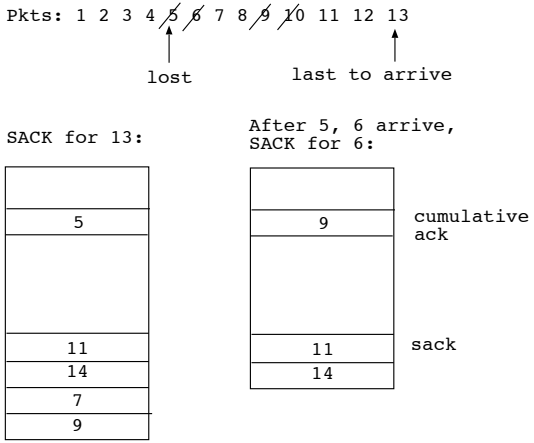
TCP then does Selective Repeat instead of Go-Back-N

Added 2 TCP options:

1. SACK Permitted (option 4): carried only by SYN packets
2. SACK Option (option 5) on TCP data packets: NAK missing segments, listed in TCP option field



TCP SACK Example



TCP SACK Advantages

Decouples when and how much to send from which to send

Potential uses of SACK info:

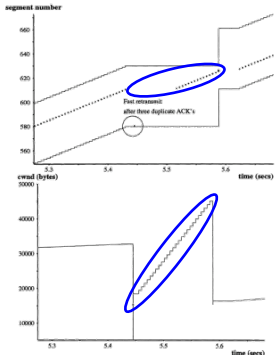
- send new packet on first or second duplicate ACKs
- tell sender when there will be no more returning ACK
- cancel previous decisions upon updated info
- construct a better count of outstanding segments

Deployed since Windows98, Linux 2.6, and also in Mac OS X

Rate Halving

Upon loss detection, instead of resuming transmission only after half the expected number of ACKs have returned, send one segment for every 2 segment ACKed

- pacing out packets reduces loss
- works with SACK to estimate number of in-flight segments
- segment sent can be new or retransmission based on SACK
- partial ACK doesn't take TCP out of fast retransmit phase



Proportional Rate Reduction (PRR)

PRR adopted as default TCP loss recovery mechanism in Linux since v. 3.2 (2012)

Upon loss detection, instead of always halving the sending rate while in recovery mode, reduce the rate proportionally to the new `cwnd` (=`ssthresh`) size

Proportional Rate Reduction (PRR)

With Reno, the rate is simply halved:

Upon loss:

```
pipe = cwnd;
ssthresh = cwnd/2;
```

Upon every ACK or SACK:

```
// update delivered and outstanding
cwnd = outstanding + ceil(ssthresh/pipe
       * delivered) - recovery_sent;
```

New packet sent out if `cwnd` > `outstanding`

```
// update recovery_sent
```

With CUBIC, new `cwnd` is roughly a third of old `cwnd`

Fairness Issues

How do flows with different RTTs share link?

- how would different RTTs affect fairness?

UDP

- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - pump audio/video data at constant rate, tolerate packet losses
- TCP-friendly UDP?

Fairness Issues

Parallel TCP connections

- nothing prevents app from opening parallel connections between 2 hosts
  - web browsers do this already
- Example: link of rate R currently supporting 9 connections
  - new app opens 1 TCP, gets $R/10$
  - new app opens 11 TCPs, gets $R/2$!

Problem: on the Internet, there's no incentive to play fair ⇒ *Tragedy of the Commons*

Perhaps transport protocol should just optimize for minimal flow completion time

Multi-Path TCP (MP-TCP)

Allows for multiple connections (sub-flows) between a sender and a receiver, presumably using different IP paths

All the sub-flows share a single overall `cwnd` that regulates their individual `cwnd`'s

Faster paths send more traffic than slower paths, regulated by the overall `cwnd`

Supported by iOS 7 (Sept. 2013)

Quick UDP Internet Connections

Multiplexed stream transport over UDP

User-space reliable and secure stream transport (targeted to SPDY or HTTP/2)

Isn't SPDY enough?
SPDY streams multiplexed onto one TCP stream:

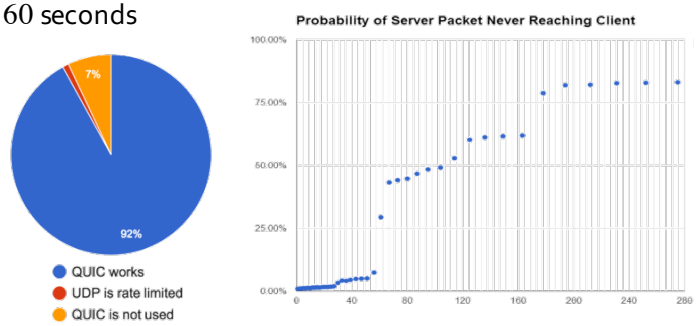
- one lost TCP segment holds up all SPDY streams
- one lost TCP segment causes bandwidth to shrink for all SPDY streams
- slow connection establishment (TCP 1 RTT, TLS +1 RTT)

[Roskind]

QUIC over NAT?

Google servers can communicate with 91-94% of users that initiates UDP-based sessions with Google

Most NAT port mapping stays bound for at least 30-60 seconds



[Google]

QUIC Features

1. Always encrypted
2. Fast Open (0-RTT)
3. Connection migration
4. Congestion control
5. FEC (XOR-based, optional)
6. Multipath (future work)

0-RTT Connection Establishment

No cached information available:

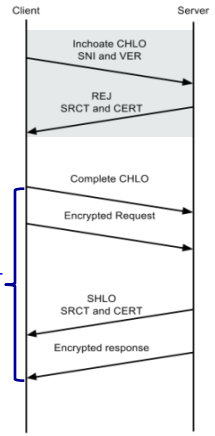
- first CHLO (client hello) is inchoate (empty): version and server name
- server responds with REJ, includes server config (token), certs, etc.

Cached information available (0-RTT):

- complete CHLO: server token, connID, etc.
  - followed by encrypted request data
- server responds with SHLO (server hello)
  - followed by encrypted response data

~75% sessions are 0-RTT, contributes

- 50-80% of median latency improvement
- 50% of 95th-%tile improvement



[Iyengar]

Connection Migration

A 64-bit connection ID, instead of the usual 4-tuple, identifies a connection

Connection ID chosen randomly by client

Enables connection mobility across IP, port

[Iyengar]

QUIC Congestion Control

Better signaling than TCP:

- each packet carries a monotonically increasing packet number
  - retransmitted packets also consume new sequence numbers (no retransmission ambiguity)
  - ACK packets consume sequence numbers
- more verbose ACK
  - support 256 NAK ranges (vs. TCP's 3 SACK ranges)

[Iyengar]

QUIC Congestion Control

Start with Linux's TCP defaults:

- TCP CUBIC
- Loss recovery: fast retransmit and PRR
- TCP SACK
- Other improvements*:
  - Forward ACK (FACK): sender uses SACK to estimate in-flight segments
  - Forward RTO Recovery (F-RTO): sender recognizes spurious RTO caused by long delay
  - Early retransmit with timer: retransmit after < 3 dupACKs if there are < 3 in-flight packets
  - Larger initial window (IW 10 to 50 packets)
  - Tail Loss Probe (TLP): always sends 2 TLPs before the first RTO

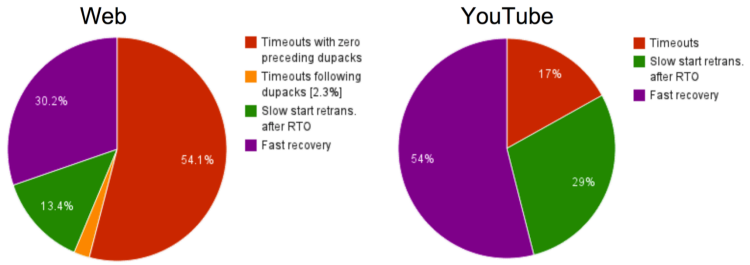
* not required to know the details of these for this course

[Swett]

Tail Loss Probe

RTOs are expensive for short flows

TCP retransmission breakdown in two Google datacenters



[Dukkipati]

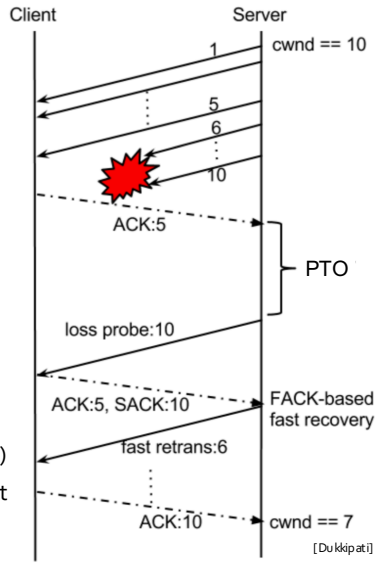
Tail Loss Probe

Observations:

- tail segments are twice more likely to be lost than earlier segments
- losses are bursty

Tail Loss Probe:

- set probe timeout (PTO) to be ~2 RTTs since last ACK received
- arriving ACK resets PTO
- upon PTO, retransmit last segment (or new one if available)
- FACK for retransmitted segment could trigger fast recovery

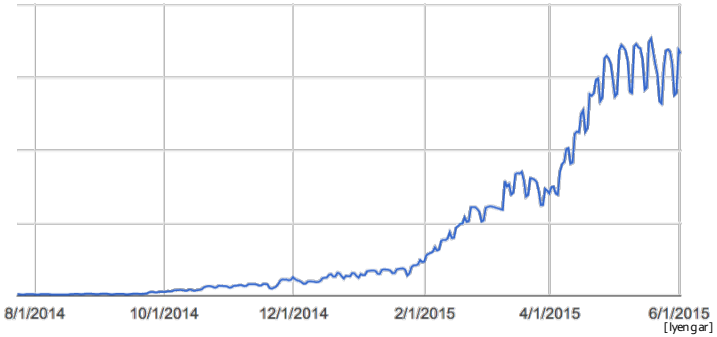


[Dukkipati]

QUIC Deployment Timeline

Tested at scale, with millions of users

- Chrome Canary: June, 2013
- Chrome Stable: April, 2014
- ramped up for Google traffic in 2015



[Iyengar]

Performance on Google Properties

Faster page loading times

- 5% faster on average
- 1 second faster for web search at 99th-percentile

0-RTT connection establishment

- over 50% of the latency improvement (at median and 95th-percentile)

Improved YouTube Quality of Experience (QoE)

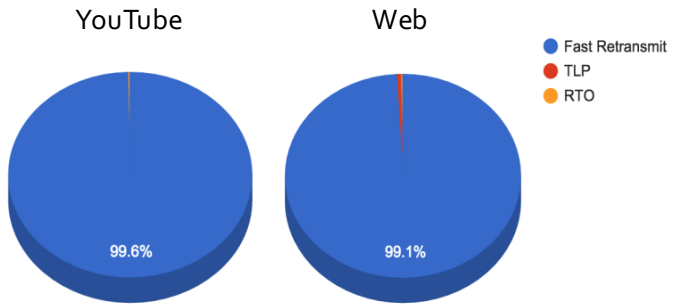
- 30% fewer rebuffers (video pauses)

Improved loss recovery

- over 10x fewer timeout based retransmissions improve tail latency and YouTube video rebuffer rates

[Swett]

QUIC Loss Recovery



[Swett]

Reno vs. CUBIC

QUIC defaults to CUBIC, similar to Linux

Latency across all services is extremely similar between Reno and CUBIC

QoE is extremely similar between Reno and CUBIC

Retransmits are ~20% lower with Reno than CUBIC

Why not default to Reno?
"*We're thinking about it…*"

[Swett]

1 vs. 2 Connection Emulation

QUIC defaults to 2-connection emulation

2-connection shows large improvements in YouTube QoE

1- vs. 2-connection has a negligible effect on median page load latency
- 2-connection shows slight improvement in tail latency

Retransmits are 20% higher with 2-connection

[Swett]

IW10 vs. IW32

QUIC defaults to 32, similar to HTTP/2 default

30% of QUIC's "time to playback" gains for YouTube due to IW32

IW10 had equal or slightly worse latency, even at the 95%

IW10 decreased retransmit rate slightly
- IW10 without pacing had higher retransmit rate than IW32 with pacing [IW considered harmful!]

(IW03, IW20, and IW50 also available)

[Swett]

Early Retransmit

QUIC defaults to FACK with a fixed dupack threshold of 3

Time-based loss detection waits ¼ RTT after the first NACK for the packet to be lost

Shows no significant improvements vs FACK on user-facing networks

[Swett]

Tail Loss Probe on QUIC

QUIC defaults to 2 TLPs before RTO

Disabling TLP has no effect on median latency

TLP improves 95% latency almost 1%

TLP improves YouTube rebuffer rate almost 1%

Disabling TLP reduces retransmits 5%

[Swett]

QUIC and BIC References

Iyengar, J., "QUIC Redefining Internet Transport," IETF-93, July 2015

Swett, I., "QUIC Congestion Control and Loss Recovery," IETF-93, July 2015

Rochkind, J., "QUIC Multiplexed Stream Transport over UDP," IETF-88, Nov. 2013

Dukkipati, N., "Tail Loss Probe (TLP)," IETF-84, Aug. 2012

Rhee, I., "Congestion Control on High-Speed Networks," IEEE Infocom, 2004

[Swett]