BBR Update:
1: BBR.Swift; 2: Scalable Loss Handling

TCP BBR, BBR.Swift: Neal Cardwell, Yuchung Cheng, Kevin Yang

Soheil Hassas Yeganeh, Priyaranjan Jha, Yousuk Seung, Luke Hsiao, Matt Mathis

Van Jacobson

QUIC BBR: Ian Swett, Bin Wu, Victor Vasiliev

Swift: Nandita Dukkipati, Gautam Kumar

https://groups.google.com/d/forum/bbr-dev

IETF 109: Online, Nov 2020

Outline

- BBR.Swift: using delay as a signal in the datacenter
- Scalable loss handling: BBR, PRR, and the scalability of multiplicative decrease
- Status of the BBR code and Google deployment
- Conclusion

Target for this talk:

- Sharing our experience with experiments
- Inviting the community share feedback, test results, issues, patches, or ideas

BBR.Swift:
Delay as a signal in the datacenter

Swift Congestion Control

- Swift congestion control algorithm [SIGCOMM-2020]
  - Uses network RTT and host delays as primary signals (also uses loss)
  - Potential scope:
    - Inside network with known topology/RTT properties (e.g. datacenters)
    - With NICs with hardware timestamp support
    - Where all traffic sharing bottleneck is Swift-compatible
  - Algorithm:
    - AIMD, with MD (multiplicative decrease) proportional to the excess delay
    - Uses pacing when cwnd < 1, to handle large-scale multiplexing (e.g. "incast")
  - Usage so far:
    - In Snap userspace network stack [SOSP-2019]
    - Traffic within a Google datacenter
      - Target network RTT is known
      - Known that other traffic sharing QoS queue is Swift-compatible

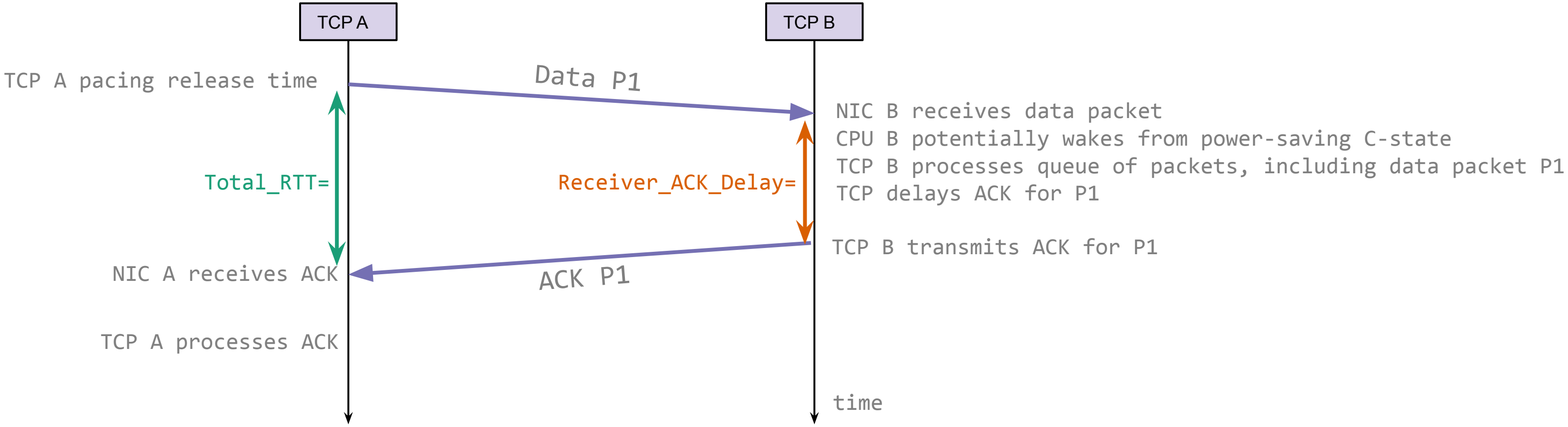
Swift: Delay as a Congestion Signal

- Advantages of using delay as a signal:
    - Provides richer information about the current degree of queuing
        - Allows faster reaction to long queues
        - Avoids overreaction and underutilization with short queues
    - Provides a known target latency for engineering systems
        - Helps applications to predict the latency they should expect
        - Helps set SLOs of network queuing delay for engineering, monitoring, alerting
        - By contrast, loss rate or ECN monitoring is difficult to translate into application performance (e.g. how does x% loss relate to latency?)

- Key requirement: *accurate* delay measurements of network and host

BBR.Swift: Network_RTT Signal

- Data sender computes a Network_RTT sample using:
  - Network_RTT = Total_RTT - Receiver_ACK_Delay



- Data and ACK transmission times: measured by TCP
- Data and ACK reception times: measured by NIC (via NIC hardware rx timestamps)
- Data receiver signals Receiver_ACK_Delay using new timestamp option (draft-yang-tcpm-ets-00, IETF 109 TCPM)

BBR.Swift: Design

- BBR.Swift is an extension of BBRv2
  - Core BBRv2 unchanged
  - Extension to BBRv2 based on Swift [SIGCOMM-2020]
  - New configuration parameter: target_RTT (e.g. O(100us) inside DC)
  - If Network_RTT > target_RTT, multiply cwnd and pacing rate by:

$$\text{MD} = \max\left(1 - 0.8 * \frac{\text{Network\_RTT} - \text{target\_RTT}}{\text{Network\_RTT}}, 0.5\right)$$

- DCTCP-style ECN response is disabled if target_RTT is used
  - For interactions between BBR.ECN WAN flows vs BBR.Swift flows:
    - Exploring ideas, e.g.: dynamically set target_RTT to the Network_RTT at the boundary of ECN marking

BBR.Swift: Example Performance Results in the Lab

- 3 server-class machines w/ 50G NICs on same switch; controlled lab network setting
- 2 senders, 1 receiver; 2000 bulk TCP flows;
- Each sender sends 1000 bulk netperf TCP_STREAM flows for 10 secs
  - Second sender starts ~0.2 sec after first sender
- Results reported for each data sender machine

- DCTCP issues:
  - Floor is cwnd=1
  - Big standing queue
  - Large loss rate
  - Less fair
- BBR.Swift features:
  - Pacing ~= delivery rate
  - Small queue
  - Low loss rate
  - Quite fair

Congestion Control	DCTCP		BBRv2 (ECN)		BBR.Swift	
Sender machine	1	2	1	2	1	2
Throughput (Gbit/sec)	46.6	3.7	25.0	23.5	24.9	25.4
Mean Total_RTT (us)			228	337	196	195
Mean Network_RTT (us)					93	93
Retransmit rate	6.1%	66.2%	1.6%	1.7%	0.053%	0.053%
Jain's fairness index	0.671	0.746	0.69	0.70	0.956	0.957

BBR.Swift: Status and Plans

- Preparing for production testing of BBR.Swift for Google internal traffic
- Plan to release BBR.Swift code as open source and document the algorithm in detail
    - Including implementation of draft-yang-tcpm-ets-00
- Goal: we want transports to be able to use BBR.Swift as their CC
    - On connections where...
        - Target Network_RTT is known
        - It is known that other traffic sharing bottlenecks is using Swift or BBR.Swift
    - On physical machines or virtual machines

Congestion Control in Loss Recovery

Multiplicative Decrease is Not Scalable Enough

- Traditional TCP CC uses multiplicative decrease upon round trips with packet loss
  - e.g., Reno: 0.5x per round; CUBIC: 0.7x per round
- But what if the bandwidth available to a flow suddenly drops by 1000x?
  - Theory: Reno expects log_2(old_bw / new_bw) round trips of high loss rate pain
    - e.g. 1000x cut in fair-share bandwidth can lead to 10 rounds of high loss
  - Reality:
    - With TCP loss recovery before RACK, consecutive rounds of loss => RTO
      - Before RACK, lost retransmits usually caused RTO, cwnd=1, slow-start
    - With TCP RACK but no PRR, reality matches theory
      - And the resulting high loss rate can be painful!
    - With TCP RACK and PRR, sending rate is bounded to be near delivery rate
      - And the loss rate is reasonable and tolerable

How Should BBR Respond to Loss?

- BBRv1 used an approach inspired by PRR
    - First round of recovery: packet conservation
    - Subsequent rounds of recovery: send at 2x the per-packet delivery rate
- Initial revisions of BBRv2 simplified things by using a multiplicative decrease
    - Similar to 0.7x per round used by CUBIC
- Production experience with Google-internal datacenter RPC traffic shows initial BBRv2...
    - Approaches the painful theoretical behavior: high loss for log(old_bw / new_bw)
- Next: experimenting with various PRR-inspired responses to loss recovery for BBRv2
    - Stay tuned for production experiment results...

Wrapping up...

Status of BBR v2 algorithm and code

- TCP BBRv2 "alpha/preview" release:
  - Linux TCP (dual GPLv2/BSD): **github.com/google/bbr/blob/v2alpha/README.md**
- QUIC BBR v2:
  - Chromium QUIC (BSD):on chromium.org in bbr2_sender.{ cc, h }

- BBR v2 release is ready for research experiments
  - We invite researchers to share...
    - Ideas for test cases and metrics to evaluate
    - Test results
    - Algorithm/code ideas
  - Always happy to see patches or look at packet traces...
- BBR v2 algorithm was described at IETF 104 [slides | video]
- BBR v2 open source release was described at IETF 105 [slides | video]

BBR v2 deployment status at Google

- YouTube and google.com: deployed for a small percentage of users
    - Reduced queuing delays: RTTs lower than BBR v1 and CUBIC
    - Reduced packet loss: loss rates closer to CUBIC than BBR v1
- Google-internal traffic:
    - BBRv2 being deployed as default TCP congestion control for internal Google traffic
    - Used as the congestion control for most traffic within Google
        - Currently using bandwidth * min_rtt, loss, ECN as signals
        - Preparing for production testing using Network_RTT
- Continuing to iterate using production experiments and lab tests

Conclusion

- Actively working on BBR v2, BBR.Swift at Google
    - Tuning performance to enable full-scale roll-out at Google
    - Improving the algorithm to scale to larger numbers of flows
    - We invite the community share test results, issues, patches, or ideas

Q & A

https://groups.google.com/d/forum/bbr-dev

Internet Drafts, paper, code, mailing list, talks, etc.

Special thanks to Eric Dumazet, Nandita Dukkipati, C. Stephen Gunn, Jana Iyengar, Pawel Jurczyk, Biren Roy, David Wetherall, Amin Vahdat, Leonidas Kontothanassis, and {YouTube, google.com, SRE, BWE} teams.

Backup slides...

ECN: Responsiveness vs Utilization Trade-Offs

- [RFC3168]-style ECN:
    - Coarse once-per-RTT signal forces high queues or low utilization
- DCTCP/Prague-style ECN:
    - Per-ACK ECN signal is much better than [RFC3168], but still has a dilemma...
    - How to choose the magnitude of the multiplicative decrease (MD)?
    - MD based on EWMA of ECN mark rate (DCTCP, TCP Prague, BBRv2):
        - Delays response to suddenly long queues
    - MD using instantaneous ECN mark rate in current round trip (GCN):
        - Allows fast reaction to long/congested queues
        - But if queue is actually short or RTT is long, a big MD causes underutilization
- Ideally we'd like:
    - An MD proportional to a per-ACK signal that quantifies the magnitude of the queue
    - Adapts to use a large MD to drain long queues quickly
    - Adapts to use a small MD to allow high utilization even for short queues

BBR.Swift: Network_RTT via ETS timestamps

- Extensible Timestamp option (ETS): draft-yang-tcpm-ets-00 (see TCPM at IETF 109)
  - Has TSval, TSecr, like [RFC7323] timestamp option, but...
  - Explicitly signals Receiver_ACK_Delay
  - TSval, TSecr, Receiver_ACK_Delay are all in microsecond granularity
- Data sender computes Total_RTT using ETS timestamp options:
  - Total_RTT = (time NIC received ACK) - (scheduled pacing release time of data)
- Data receiver computes Receiver_ACK_Delay:
  - Receiver_ACK_Delay = (scheduled release time of ACK) - (time NIC received data)
  - Data receiver sends Receiver_ACK_Delay using ETS timestamp option

BBR.Swift: ETS Timestamp Option Header Format

```
        +--------+--------+--------+--------+
        |  Kind  | Length |      ExID       |
        +--------+--------+--------+--------+
        |               TSval               |
        +--------+--------+--------+--------+
        |               TSecr               |
        +--------+--------+--------+--------+
        |Un|   EcrDel   |R|    MaxACKDel    |
        +---------------+-+-----------------+
        |                 |
       /                   \
  .---'                     `---------------------.
 /                                                 \
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Unit  |            EcrDel               |Reserved|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  2 bits |            13 bits              | 1 bit
```

Kind: 1 byte, value 254, [RFC6994] experimental option
Length: 1 byte option length, value is 16 if SYN bit is set,
otherwise 14 (value MAY be higher in later versions).
ExID: 2 bytes, [RFC6994] experiment ID: value 0x4554.
TSval and TSecr: 32 bits each, have the same definition as
[RFC7323] but are in **microseconds**.
EcrDelUnit: 2 bits; allowed values are:
0: indicates EcrDel is in microsecond units
1: indicates EcrDel is in millisecond units
2: indicates EcrDel is invalid (should be ignored)
3: reserved in this protocol version
EcrDel: 13 bits, the value of EcrDel.
Reserved: 1 bit, in this protocol version, sender MUST set to 0
And receiver MUST ignore.
MaxACKDel: 16 bits, max expected ACK delay in microseconds,
only present in SYN.