Internet Congestion Control Research Group
Internet-Draft
Intended status: Experimental
Expires: January 4, 2018

Y. Cheng
N. Cardwell
S. Hassas Yeganeh
V. Jacobson
Google, Inc

July 03, 2017

Delivery Rate Estimation
draft-cheng-iccrg-delivery-rate-estimation-00

Abstract

This document describes a generic algorithm for a transport protocol sender to estimate the current delivery rate of its data. At a high level, the algorithm estimates the rate at which the network delivered the most recent flight of outbound data packets for a single flow. In addition, it tracks whether the rate sample was application-limited, meaning the transmission rate was limited by the application rather than the congestion control algorithm. This algorithm can be implemented in any transport protocol that supports packet-delivery acknowledgment (thus far, open source implementations are available for TCP [RFC793] and QUIC [draft-ietf-quic-transport-00]).

Table of Contents

1. Introduction 3
2. Algorithm Overview 3
2.1. Requirements 3
2.2. Estimating Delivery Rate 3
2.2.1. ACK Rate 4
2.2.2. ACK Compression and Aggregation 5
2.2.3. Send Rate 5
2.2.4. Delivery Rate 6
2.3. Tracking application-limited phases 6
3. Detailed Algorithm 7
3.1. Variables 7
3.1.1. Per-connection (C) state 7
3.1.2. Per-packet (P) state 8
3.1.3. Rate Sample (rs) Output 8
3.2. Transmitting or retransmitting a data packet 9
3.3. Upon receiving an ACK 10
3.4. Detecting application-limited phases 11
4. Discussion 11
4.1. Offload Mechanisms 11
4.2. Impact of ACK losses 12
4.3. Impact of packet reordering 12
4.4. Impact of packet loss and retransmissions 12
4.5. Connections without SACK support 13
5. Implementation Status 13
6. Security Considerations 14
7. IANA Considerations 14
8. Acknowledgments 14
9. References 15
9.1. Normative References 15
9.2. Informative References 15
Authors' Addresses 16

1. Introduction

This document describes a generic algorithm for a transport protocol sender to estimate the current delivery rate of its data on the fly. This technique has been used for a congestion control algorithm that relies on fresh, reliable, and inexpensive delivery rate information [draft-cardwell-iccrg-bbr-congestion-control] [CCGHJ17].

At a high level, the algorithm estimates the rate at which the network delivered the most recent flight of outbound data packets for a single flow. In addition, it tracks whether the rate sample was application-limited, meaning the transmission rate was limited by the application rather than the congestion control algorithm.

Each acknowledgment that cumulatively or selectively acknowledges that the network has delivered new data produces a rate sample which records the amount of data delivered over the time interval between the transmission of a data packet and the acknowledgment of that packet. The samples reflect the recent goodput through some bottleneck, which may reside either in the network or on the end hosts (sender or receiver).

2. Algorithm Overview

2.1. Requirements

This algorithm can be implemented in any transport protocol that supports packet-delivery acknowledgment (so far, implementations are available for TCP [RFC793] and QUIC [draft-ietf-quic-transport-00]). This algorithm requires a small amount of added logic on the sender, and requires that the sender maintain a small amount of additional per-packet state for packets sent but not yet delivered. In the most general case it requires high-precision (microsecond-granularity or better) timestamps on the sender (though millisecond-granularity may suffice for lower bandwidths). It does not require any receiver or network changes. While selective acknowledgments for out-of-order data (e.g., [RFC2018]) are not required, such a mechanism is highly recommended for accurate estimation during reordering and loss recovery phases.

2.2. Estimating Delivery Rate

A delivery rate sample records the estimated rate at which the network delivered packets for a single flow, calculated over the time interval between the transmission of a data packet and the acknowledgment of that packet. Since the rate samples only include packets actually cumulatively and/or selectively acknowledged, the sender knows the exact octets that were delivered to the receiver

(not lost), and the sender can can compute an estimate of a bottleneck delivery rate over that time interval.

The amount of data delivered MAY be tracked in units of either octets or packets. Tracking data in units of octets is more accurate, since packet sizes can vary. But for some purposes, including congestion control, tracking data in units of packets may suffice.

2.2.1. ACK Rate

First, consider the rate at which data is acknowledged by the receiver. In this algorithm, the computation of the ACK rate models the average slope of a hypothetical "delivered" curve that tracks the cumulative quantity of data delivered so far on the Y axis, and time elapsed on the X axis. Since ACKs arrive in discrete events, this "delivered" curve forms a step function, where each ACK causes a discrete increase in the "delivered" count that causes a vertical upward step up in the curve. This "ack_rate" computation is the average slope of the "delivered" step function, as measured from the "knee" of the step (ACK) preceding the transmit to the "knee" of the step (ACK) for packet P.

Given this model, the ack rate sample "slope" is computed as the ratio between the amount of data marked as delivered over this time interval, and the time over which it is marked as delivered:

```
ack_rate = data_acked / ack_elapsed
```

To calculate the amount of data ACKed over the interval, the sender records in per-packet state "P.delivered", the amount of data that had been marked delivered before transmitting packet P, and then records how much data had been marked delivered by the time the ACK for the packet arrives (in "C.delivered"), and computes the difference:

```
data_acked = C.delivered - P.delivered
```

To compute the time interval, "ack_elapsed", one might imagine that it would be feasible to use the round-trip time (RTT) of the packet. But it is not safe to simply calculate a bandwidth estimate by using the time between the transmit of a packet and the acknowledgment of that packet. Transmits and ACKs can happen out of phase with each other, clocked in separate processes. In general transmits often happen at some point later than the most recent ACK, due to processing or pacing delays. Because of this effect, drastic over-estimates can happen if a sender were to attempt to estimate bandwidth by using the round-trip time.

This document specifies the following approach for computing "ack_elapsed". The starting time is "P.delivered_time", the time of the delivery curve "knee" from the ACK preceding the transmit. The ending time is "C.delivered_time", the time of the delivery curve "knee" from the ACK for P. Then we compute "ack_elapsed" as:

```
ack_elapsed = C.delivered_time - P.delivered_time
```

This yields our equation for computing the ACK rate, as the "slope" from the "knee" preceding the transmit to the "knee" at ACK:

```
ack_rate = data_acked / ack_elapsed
ack_rate = (C.delivered - P.delivered) /
           (C.delivered_time - P.delivered_time)
```

2.2.2. ACK Compression and Aggregation

For computing the delivery_rate, the sender prefers ack_rate, the rate at which packets were acknowledged, since this usually the most reliable metric. However, this approach of directly using "ack_rate" faces a challenge when used with paths featuring ACK decimation, aggregation, or compression, which are prevalent [A15]. In such cases, ACK arrivals can temporarily make it appear as if data packets were delivered much faster than the bottleneck rate. To filter out such implausible ack_rate samples, we consider the send rate for each flight of data, as follows.

2.2.3. Send Rate

The sender calculates the send rate, "send_rate", for a flight of data as follows. Define "P.first_sent_time" as the time of the first send in a flight of data, and "P.sent_time" as the time the final send in that flight of data (the send that transmits packet "P"). The elapsed time for sending the flight is:

```
send_elapsed = (P.sent_time - P.first_sent_time)
```

Then we calculate the send_rate as:

```
send_rate = data_acked / send_elapsed
```

Using our "delivery" curve model above, the send_rate can be viewed as the average slope of a "send" curve that traces the amount of data sent on the Y axis, and the time elapsed on the X axis: the average slope of the transmission of this flight of data.

2.2.4. Delivery Rate

Since it is physically impossible to have data delivered faster than it is sent in a sustained fashion, when the estimator notices that the ack_rate for a flight is faster than the send rate for the flight, it filters out the implausible ack_rate by capping the delivery rate sample to be no higher than the send rate.

More precisely, over the interval between each transmission and corresponding ACK, the sender calculates a delivery rate sample, "delivery_rate", using the minimum of the rate at which packets were acknowledged or the rate at which they were sent:

```
delivery_rate = min(send_rate, ack_rate)
```

Since ack_rate and send_rate both have data_acked as a numerator, this can be computed more efficiently with a single division (instead of two), as follows:

```
delivery_elapsed = max(ack_elapsed, send_elapsed)
delivery_rate = data_acked / delivery_elapsed
```

2.3. Tracking application-limited phases

In application-limited phases the transmission rate iss limited by the application rather than the congestion control algorithm. Modern transport protocol connections are often application-limited, either due to request/response workloads (e.g. Web traffic, RPC traffic) or because the sender transmits data in chunks (e.g. adaptive streaming video).

Knowing whether a delivery rate sample was application-limited is crucial for congestion control algorithms and applications to use the estimated delivery rate samples properly. For example, congestion control algorithms may not want to react to a delivery rate that is lower simply because the sender is application-limited; for congestion control the key metric is the rate at which the network path delivers data, and not simply the rate at which the application happens to be transmitting data at any moment.

To track this, the estimator marks a bandwidth sample as application-limited if there was some moment during the sampled window of data packets when there was no data ready to send.

An application-limited phase starts when the sending application requests to send more data and meets all of the following conditions

1. The transport send buffer has less than one SMSS of unsent data available to send.

2. The sending flow is not currently in the process of transmitting a packet.

3. The amount of data considered in flight is less than the congestion window (cwnd).

4. All the packets considered lost have been retransmitted.

If these conditions are all met then the sender has run out of data to feed the network. This would effectively create a "bubble" of idle time in the data pipeline. This idle time means that any delivery rate sample obtained from this data packet, and any rate sample from a packet that follows it in the next round trip, is going to be an application-limited sample that potentially underestimates the true available bandwidth. Thus, when the algorithm marks a transport flow as application-limited, it marks all bandwidth samples for the next round trip as application-limited (at which point, the "bubble" can be said to have exited the data pipeline).

3. Detailed Algorithm

3.1. Variables

3.1.1. Per-connection (C) state

This algorithm requires the following new state variables for each transport connection:

C.delivered: The total amount of data (tracked in octets or in packets) delivered so far over the lifetime of the transport connection.

C.delivered_time: The wall clock time when C.delivered was last updated.

C.first_sent_time: If packets are in flight, then this holds the send time of the packet that was most recently marked as delivered. Else, if the connection was recently idle, then this holds the send time of most recently sent packet.

C.app_limited: The index of the last transmitted packet marked as application-limited, or 0 if the connection is not currently application-limited.

We also assume that the transport protocol sender implementation tracks the following state per connection. If the following state variables are not tracked by an existing implementation, all the following parameters MUST be tracked to implement this algorithm:

C.write_seq: The data sequence number one higher than that of the last octet queued for transmission in the transport layer write buffer.

C.pending_transmissions: The number of bytes queued for transmission on the sending host at layers lower than the transport layer (i.e. network layer, traffic shaping layer, network device layer).

C.lost_out: The number of packets in the current outstanding window that are marked as lost.

C.retrans_out: The number of packets in the current outstanding window that are being retransmitted.

C.pipe: The sender’s estimate of the number of packets outstanding in the network; i.e. the number of packets in the current outstanding window that are being transmitted or retransmitted and have not been SACKed or marked lost (e.g. "pipe" from [RFC6675]).

3.1.2. Per-packet (P) state

This algorithm requires the following new state variables for each packet that has been transmitted but not yet ACKed or SACKed:

P.delivered: C.delivered at the time the packet was sent.

P.delivered_time: C.delivered_time at the time the packet was sent.

P.first_sent_time: C.first_sent_time at the time the packet was sent.

P.is_app_limited: C.app_limited at the time the packet was sent.

P.sent_time: The time when the packet was sent.

3.1.3. Rate Sample (rs) Output

This algorithm provides its output in a RateSample structure rs, containing the following fields:

rs.delivery_rate: The delivery rate sample (in most cases rs.delivered / rs.interval).

rs.is_app_limited: The P.is_app_limited from the most recent packet delivered; indicates whether the rate sample is application-limited.

rs.interval: The length of the sampling interval.

rs.delivered: The amount of data marked as delivered over the sampling interval.

rs.prior_delivered: The P.delivered count from the most recent packet delivered.

rs.prior_time: The P.delivered_time from the most recent packet delivered.

rs.send_elapsed: Send time interval calculated from the most recent packet delivered (see the "Send Rate" section above).

rs.ack_elapsed: ACK time interval calculated from the most recent packet delivered (see the "ACK Rate" section above).

3.2. Transmitting or retransmitting a data packet

Upon transmitting or retransmitting a data packet, the sender snapshots the current delivery information in per-packet state. This will allow the sender to generate a rate sample later, in the UpdateRateSample() step, when the packet is (S)ACKed.

If there are packets already in flight, then we need to start delivery rate samples from the time we received the most recent ACK, to try to ensure that we include the full time the network needs to deliver all in-flight packets. If there are no packets in flight yet, then we can start the delivery rate interval at the current time, since we know that any ACKs after now indicate that the network was able to deliver those packets completely in the sampling interval between now and the next ACK.

Upon each packet transmission, the sender executes the following steps:

```
  SendPacket(Packet P):
    if (C.pipe == 0)
      C.first_sent_time  = C.delivered_time = Now()
    P.first_sent_time = C.first_sent_time
    P.delivered_time  = C.delivered_time
    P.delivered       = C.delivered
    P.is_app_limited  = (C.app_limited != 0)
```

3.3. Upon receiving an ACK

When an ACK arrives, the sender invokes GenerateRateSample() to fill in a rate sample. For each packet that was newly SACKed or ACKed, UpdateRateSample() updates the rate sample based on a snapshot of connection delivery information from the time at which the packet was last transmitted. UpdateRateSample() is invoked multiple times when a stretched ACK acknowledges multiple data packets. In this case we use the information from the most recently sent packet, i.e., the packet with the highest "P.delivered" value.

```
  /* Upon receiving ACK, fill in delivery rate sample rs. */
  GenerateRateSample(RateSample rs):
    for each newly SACKed or ACKed packet P
      UpdateRateSample(P, rs)

    /* Clear app-limited field if bubble is ACKed and gone. */
    if (C.app_limited and C.delivered > C.app_limited)
      C.app_limited = 0

    if (rs.prior_time == 0)
      return false  /* nothing delivered on this ACK */

    /* Use the longer of the send_elapsed and ack_elapsed */
    rs.interval = max(rs.send_elapsed, rs.ack_elapsed)

    rs.delivered = C.delivered - rs.prior_delivered

    /* Normally we expect interval >= MinRTT.
     * Note that rate may still be over-estimated when a spuriously
     * retransmitted skb was first (s)acked because "interval"
     * is under-estimated (up to an RTT). However, continuously
     * measuring the delivery rate during loss recovery is crucial
     * for connections suffer heavy or prolonged losses.
     */
    if (rs.interval <  MinRTT(tp))
      rs.interval = -1
      return false  /* no reliable sample */

    if (rs.interval != 0)
      rs.delivery_rate = rs.delivered / rs.interval

    return true;  /* we filled in rs with a rate sample */

  /* Update rs when packet is SACKed or ACKed. */
  UpdateRateSample(Packet P, RateSample rs):
    if P.delivered_time == 0
      return /* P already SACKed */
```

```
      C.delivered += P.data_length
      C.delivered_time = Now()

      /* Update info using the newest packet: */
      if (P.delivered > rs.prior_delivered)
        rs.prior_delivered  = P.delivered
        rs.prior_time       = P.delivered_time
        rs.is_app_limited   = P.is_app_limited
        rs.send_elapsed     = P.sent_time - P.first_sent_time
        rs.ack_elapsed      = C.delivered_time - P.delivered_time
        C.first_sent_time   = P.sent_time

      /* Mark the packet as delivered once it's SACKed to
       * avoid being used again when it's cumulatively acked.
       */
      P.delivered_time = 0
```

3.4. Detecting application-limited phases

An application-limited phase starts when the sending application asks the transport layer to send more data and the connection has run out of data. Upon each write from the application, the algorithm checks all of the conditions previously described in the "Tracking application-limited phases" section, and if all are met then it marks the connection as application-limited:

```
  /* On gaps between sends, mark flow application-limited: */
  OnApplicationWrite():
    if (C.write_seq - SND.NXT < SND.MSS and
        C.pending_transmissions == 0 and
        C.pipe < cwnd and
        C.lost_out <= C.retrans_out)
      C.app_limited = C.delivered + C.pipe ? : 1
```

4. Discussion

4.1. Offload Mechanisms

If a transport sender implementation uses an offload mechanism (such as TSO, GSO, etc.) to combine multiple SMSS of data into a single packet "aggregate" for the purposes of scheduling transmissions, then it is RECOMMENDED that the per-packet state be tracked for each packet "aggregate" rather than each SMSS. For simplicity this document refers to such state as "per-packet", whether it is per "aggregate" or per SMSS.

4.2. Impact of ACK losses

Delivery rate samples are generated upon receiving each ACK; ACKs may contain both cumulative and selective acknowledgment information. Losing an ACK results in losing the delivery rate sample corresponding to that ACK, and generating a delivery rate sample at later a time (upon the arrival of the next ACK). This can underestimate the delivery rate due the artificially inflated "rs.interval". As with any effect that can cause underestimation, it is RECOMMENDED that applications or congestion control algorithms using the output of this algorithm apply appropriate filtering to mitigate the impact of this effect.

4.3. Impact of packet reordering

This algorithm is robust to packet reordering; it makes no assumptions about the order in which packets are delivered or ACKed. In particular, for a particular packet P, it does not matter which packets are delivered between the transmission of P and the ACK of packet P, since C.delivered will be incremented appropriately in any case.

4.4. Impact of packet loss and retransmissions

There are several possible approaches for handling cases where a delivery rate sample is based on an ACK or SACK for a retransmitted packet.

If the transport protocol supports unambiguous ACKs for retransmitted data sequence ranges (as in QUIC [draft-ietf-quic-transport-00]) then the algorithm is perfectly robust to retransmissions, because the starting packet, P, for the sample can be unambiguously retrieved.

If the transport protocol, like TCP [RFC793], has ambiguous ACKs for retransmitted sequence ranges, then the following approaches MAY be used:

1. The sender MAY choose to filter out implausible delivery rate samples, as described in the GenerateRateSample() step in the "Upon receiving an ACK" section, by discarding samples whose rs.interval is lower than the minimum RTT seen on the connection.

2. The sender MAY choose to skip the generation of a delivery rate sample for a retransmitted sequence range.

4.5. Connections without SACK support

If the transport connection does not use SACK (i.e., either or both ends of the connections do not accept SACK), then this algorithm can be extended to estimate approximate delivery rates using duplicate ACKs (much like Reno and [RFC5681] estimates that each duplicate ACK indicates that a data segment has been delivered). The details of this extension will be described in a future version of this draft.

5. Implementation Status

This section records the status of known implementations of the algorithm defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC7942]. The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs. Please note that the listing of any individual implementation here does not imply endorsement by the IETF. Furthermore, no effort has been spent to verify the information presented here that was supplied by IETF contributors. This is not intended as, and must not be construed to be, a catalog of available implementations or their features. Readers are advised to note that other implementations may exist.

According to [RFC7942], "this will allow reviewers and working groups to assign due consideration to documents that have the benefit of running code, which may serve as evidence of valuable experimentation and feedback that have made the implemented protocols more mature. It is up to the individual working groups to use this information as they see fit".

As of the time of writing, the following implementations of this algorithm have been publicly released:

- Linux TCP
  - Source code URL:
    - GPLv2 license: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/ipv4/tcp_rate.c
    - BSD-style license: https://groups.google.com/d/msg/bbr-dev/X0LbDptlOzo/EVgkRjVHBQAJ
  - Source: Google
  - Maturity: production

    * License: dual-licensed: GPLv2 / BSD-style

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

    * Last updated: June 30, 2017

  o QUIC

    * Source code URLs:

      + https://chromium.googlesource.com/chromium/src/net/+/master/quic/core/congestion_control/bandwidth_sampler.cc

      + https://chromium.googlesource.com/chromium/src/net/+/master/quic/core/congestion_control/bandwidth_sampler.h

    * Source: Google

    * Maturity: production

    * License: BSD-style

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

    * Last updated: June 30, 2017

6. Security Considerations

This proposal makes no changes to the underlying security of transport protocols or congestion control algorithms. This algorithm adds no security considerations beyond those involved in the existing standard congestion control algorithm [RFC5681].

7. IANA Considerations

This document makes no request of IANA.

Note to RFC Editor: this section may be removed on publication as an RFC.

8. Acknowledgments

The authors would like to thank C. Stephen Gunn, Eric Dumazet, Ian Swett, Jana Iyengar, Victor Vasiliev, Nandita Dukkipati, Pawel Jurczyk, Biren Roy, David Wetherall, Amin Vahdat, Leonidas Kontothanassis, and the YouTube, google.com, Bandwidth Enforcer, and Google SRE teams for their invaluable help and support.

9. References

9.1. Normative References

[RFC2018] Mathis, M. and J. Mahdavi, "TCP Selective Acknowledgment Options", RFC 2018, October 1996.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, September 2009.

[RFC6675] Blanton, E., Allman, M., Wang, L., Jarvinen, I., Kojo, M., and Y. Nishida, "A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP", RFC 6675, August 2012.

[RFC793] Postel, J., "Transmission Control Protocol", September 1981.

[RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", July 2016.

9.2. Informative References

[A15] Abrahamsson, M., "TCP ACK suppression", IETF AQM mailing list , November 2015, <https://www.ietf.org/mail-archive/web/aqm/current/msg01480.html>.

[CCGHJ17] Cardwell, N., Cheng, Y., Gunn, C., Hassas Yeganeh, S., and V. Jacobson, "BBR: Congestion-Based Congestion Control", Communications of the ACM Feb 2017, February 2017.

[draft-cardwell-iccrg-bbr-congestion-control] Cardwell, N., Cheng, Y., Hassas Yeganeh, S., and V. Jacobson, "BBR Congestion Control", draft-cardwell-iccrg-bbr-congestion-control-00 (work in progress), June 2017, <https://tools.ietf.org/html/draft-cardwell-iccrg-bbr-congestion-control-00>.

[draft-ietf-quic-transport-00] Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-cheng-iccrg-delivery-rate-estimation-00 (work in progress), Nov 2016, <https://tools.ietf.org/html/draft-ietf-quic-transport-00>.

Authors' Addresses

Yuchung Cheng
Google, Inc
1600 Amphitheater Parkway
Mountain View, California 94043
USA

Email: ycheng@google.com

Neal Cardwell
Google, Inc
76 Ninth Avenue
New York, NY 10011
USA

Email: ncardwell@google.com

Soheil Hassas Yeganeh
Google, Inc
76 Ninth Avenue
New York, NY 10011
USA

Email: soheil@google.com

Van Jacobson
Google, Inc
1600 Amphitheater Parkway
Mountain View, California 94043

Email: vanj@google.com