Proportional Rate Reduction for TCP
draft-ietf-tcpm-prr-rfc6937bis-02

Abstract

   This document updates the experimental Proportional Rate Reduction
   (PRR) algorithm, described RFC 6937, to standards track. PRR
   potentially replaces the Fast Recovery to regulate the amount of data
   sent by TCP or other transport protocol during loss recovery. PRR
   accurately regulates the actual flight size through recovery such
   that at the end of recovery it will be as close as possible to the
   slow start threshold (ssthresh), as determined by the congestion
   control algorithm.

Status of This Memo

   This Internet-Draft is submitted in full conformance with the
   provisions of BCP 78 and BCP 79.

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF). Note that other groups may also distribute
   working documents as Internet-Drafts. The list of current Internet-
   Drafts is at https://datatracker.ietf.org/drafts/current/.

   Internet-Drafts are draft documents valid for a maximum of six months
   and may be updated, replaced, or obsoleted by other documents at any
   time. It is inappropriate to use Internet-Drafts as reference
   material or to cite them other than as "work in progress."

   This Internet-Draft will expire on 5 December 2022.

Copyright Notice

Table of Contents

1. Introduction

    This document updates the Proportional Rate Reduction (PRR) algorithm
    described in [RFC6937] from experimental to standards track. PRR
    accuracy regulates the amount of data sent during loss recovery, such
    that at the end of recovery the flight size will be as close as
    possible to the slow start threshold (ssthresh), as determined by the
    congestion control algorithm. PRR has been deployed in at least
    three major TCP implementations covering the vast majority of today's
    web traffic.

    The main change from RFC 6937 is the introduction of a new heuristic
    that replaces a manual configuration parameter and non-SACK support.
    The new heuristic only changes behaviors in corner cases that were
    not relevant before the Lost Retransmission Detection (LRD)
    algorithm. LRD was not implemented until after RFC 6937 was
    published. This document also includes additional discussion about
    integration into other congestion control and lost detection
    algorithms.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
"SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
document are to be interpreted as described in [RFC2119]

1.1. Document and WG Information

   Formatted: 2022-06-03 14:55:13-07:00

   Please send all comments, questions and feedback to tcpm@ietf.org

   About revision 00:

   The introduction above was drawn from draft-mathis-tcpm-rfc6937bis-
   00. All of the text below was copied verbatim from RFC 6937, to
   facilitate comparison between RFC 6937 and this document as it
   evolves.

   About revision 01:

   * Recast the RFC 6937 introduction as background

   * Made "Changes From RFC 6937" an explicit section

   * Made Relationships to other standards more explicit

   * Added a generalized safeACK heuristic

   * Provided hints for non TCP implementations

   * Added language about detecting ACK splitting, but have no advice
      on actions (yet)

   About revision 02:

   * Companion RACK loss detection RECOMMENDED

   * Non-SACK accounting in the pseudo code

   * cwnd computation in the pseudo code

* Force fast retransmit at the beginning of Fast Recovery

   * Remove deprecated Rate-Halving text

   * Fixed bugs in the example traces

2. Background

   This section is copied almost verbatim from the introduction to
   [RFC6937].

   Standard congestion control [RFC5681] requires that TCP (and other
   protocols) reduce their congestion window (cwnd) in response to
   losses. Fast Recovery, described in the same document, is the
   reference algorithm for making this adjustment. Its stated goal is
   to recover TCP's self clock by relying on returning ACKs during
   recovery to clock more data into the network. Fast Recovery
   typically adjusts the window by waiting for one half round-trip time
   (RTT) of ACKs to pass before sending any data. It is fragile because
   it cannot compensate for the implicit window reduction caused by the
   losses themselves.

   [RFC6675] makes Fast Recovery with Selective Acknowledgement (SACK)
   [RFC2018] more accurate by computing "pipe", a sender side estimate
   of the number of bytes still outstanding in the network. With
   [RFC6675], Fast Recovery is implemented by sending data as necessary
   on each ACK to prevent pipe from falling below ssthresh, the window
   size as determined by the congestion control algorithm. This
   protects Fast Recovery from timeouts in many cases where there are
   heavy losses, although not if the entire second half of the window of
   data or ACKs are lost. However, a single ACK carrying a SACK option
   that implies a large quantity of missing data can cause a step
   discontinuity in the pipe estimator, which can cause Fast Retransmit
   to send a burst of data.

   PRR avoids these excess window adjustments such that at the end of
   recovery the actual window size will be as close as possible to
   ssthresh, the window size as determined by the congestion control
   algorithm. It uses the fraction that is appropriate for the target
   window chosen by the congestion control algorithm. During PRR, one

of two additional Reduction Bound algorithms limits the total window
reduction due to all mechanisms, including transient application
stalls and the losses themselves.

We describe two slightly different Reduction Bound algorithms:
Conservative Reduction Bound (CRB), which is strictly packet
conserving; and a Slow Start Reduction Bound (SSRB), which is more
aggressive than CRB by, at most, 1 segment per ACK. PRR-CRB meets
the Strong Packet Conservation Bound described in Appendix A;
however, in real networks it does not perform as well as the
algorithms described in [RFC6675], which prove to be more aggressive
in a significant number of cases. SSRB offers a compromise by
allowing TCP to send 1 additional segment per ACK relative to CRB in
some situations. Although SSRB is less aggressive than [RFC6675]
(transmitting fewer segments or taking more time to transmit them),
it outperforms due to the lower probability of additional losses
during recovery.

The Strong Packet Conservation Bound on which PRR and both Reduction
Bounds are based is patterned after Van Jacobson's packet
conservation principle: segments delivered to the receiver are used
as the clock to trigger sending the same number of segments back into
the network. As much as possible, PRR and the Reduction Bound
algorithms rely on this self clock process, and are only slightly
affected by the accuracy of other estimators, such as pipe [RFC6675]
and cwnd. This is what gives the algorithms their precision in the
presence of events that cause uncertainty in other estimators.

The original definition of the packet conservation principle
[Jacobson88] treated packets that are presumed to be lost (e.g.,
marked as candidates for retransmission) as having left the network.
This idea is reflected in the pipe estimator defined in RFC 6675 and
used here, but it is distinct from the Strong Packet Conservation
Bound as described in Appendix A, which is defined solely on the
basis of data arriving at the receiver.

3. Changes From RFC 6937

The largest change since [RFC6937] is the introduction of a new
heuristic that uses good recovery progress (For TCP, when the latest

ACK advances snd.una and does not indicate prior fast retransmit has been lost) to select which Reduction Bound. [RFC6937] left the choice of Reduction Bound to the discretion of the implementer but recommended to use SSRB by default. For all of the environments explored in earlier PRR research, the new heuristic is consistent with the old recommendation.

The paper "An Internet-Wide Analysis of Traffic Policing" [Flach2016policing] uncovered a crucial situation not previously explored, where both Reduction Bounds perform very poorly, but for different reasons. Under many configurations, token bucket traffic policers [token_bucket] can suddenly start discarding a large fraction of the traffic when tokens are depleted, without any warning to the end systems. The transport congestion control has no opportunity to measure the token rate, and sets ssthresh based on the previously observed path performance. This value for ssthresh may cause a data rate that is substantially larger than the token rate, causing high loss. Under these conditions, both reduction bounds perform very poorly. PRR-CRB is too timid, sometimes causing very long recovery times at smaller than necessary windows, and PRR-SSRB is too aggressive, often causing many retransmissions to be lost multiple rounds. Both cases lead to prolonged recovery decimating application goodput.

Investigating these environments led to the development of a "safeACK" heuristic to dynamically switch between Reduction Bounds: by default conservatively use PRR-CRB and only switch to PRR-SSRB when ACKs indicate the recovery is making good progress (snd.una is advancing without any new losses).

This heuristic is only invoked where application-limited behavior, losses or other events cause the flight size to fall below ssthresh. The extreme loss rates that make the heuristic essential are only common in the presence of heavy losses such as traffic policers [Flach2016policing]. In these environments the heuristic serves to salvage a bad situation and any reasonable implementation of the heuristic performs far better than either bound by itself.

Another subtle change is to force a fast retransmit upon the first

ACK that triggers the recovery. Previously PRR may not allow a fast
retransmit (i.e. sndcnt is 0) on the first ACK depending on the loss
situation. Forcing a fast retransmit is important to keep the ACK
clock and avoid potential RTO events. The forced fast retransmit
only happens once during the entire recovery and still follows the
packet conservation principles in PRR. This heuristic has been in
the first widely deployed implementation.

Since [RFC6937] was written, PRR has also been adapted to perform
multiplicative window reduction for non-loss based congestion control
algorithms, such as for [RFC3168] style ECN. This is typically done
by using some parts of the loss recovery state machine (in particular
the RecoveryPoint from [RFC6675]) to invoke the PRR ACK processing
for exactly one round trip worth of ACKs.

For [RFC6937] we published a companion paper [IMC11] in which we
evaluated [RFC3517] and various experimental PRR versions in a large
scale measurement study. Today, the legacy algorithms used in that
study have already faded from the code base, making such comparisons
impossible without recreating historical algorithms. Readers
interested in the measurement study should review section 5 of RFC
6937 and the IMC paper [IMC11].

4. Relationships to other standards

PRR is described as modifications to "TCP Congestion Control"
[RFC5681], and "A Conservative Loss Recovery Algorithm Based on
Selective Acknowledgment (SACK) for TCP" [RFC6675]. It is most
accurate with SACK [RFC2018] but does not require SACK.

The SafeACK heuristic came about as a result of robust Lost
Retransmission Detection under development in an early precursor to
[RFC8985]. Without LRD, policers that cause very high loss rates are

Mathis, et al.   Expires 5 December 2022   [Page 6]

guaranteed to also cause retransmission timeouts because both
[RFC5681] and [RFC6675] will send retransmissions above the policed
rate. It is RECOMMENDED that PRR is implemented together with
[RFC8985].

5. Definitions

The following terms, parameters, and state variables are used as they

are defined in earlier documents:

[RFC 793](): snd.una (send unacknowledged).

[RFC 5681](): duplicate ACK, FlightSize, Sender Maximum Segment Size (SMSS).

[RFC 6675](): covered (as in "covered sequence numbers").

Voluntary window reductions: choosing not to send data in response to some ACKs, for the purpose of reducing the sending window size and data rate.

PRR defines additional variables:

DeliveredData is the number of bytes newly delivered by the most recent ACK. With SACK, DeliveredData is the change in snd.una plus the bytes newly selectively acknowledged. Without SACK, DeliveredData is the change in snd.una for a partial ACK or 1 MSS worth of bytes for a DUPACK.

Note that without SACK, a poorly-behaved receiver that returns extraneous DUPACKs as depicted in [Savage99] can artificially inflate DeliveredData. As a mitigation, PRR disallows incrementing DeliveredData when the total bytes delivered exceeds the outstanding data upon recovery (i.e., RecoverFS).

safeACK: A local boolean variable indicating that the current ACK reported good progress. SafeACK is true only when the ACK has cumulatively acknowledged new data and the ACK does not indicate further losses. Both conditions indicate the recovery is making good progress and can send more aggressively.

sndcnt: A local variable indicating exactly how many bytes should be sent in response to each ACK. Note that the decision of which data to send (e.g., retransmit missing data or send more new data) is out of scope for this document.

6. Algorithms

At the beginning of recovery, initialize the PRR state. This assumes
a modern congestion control algorithm, CongCtrlAlg(), that might set
ssthresh to something other than FlightSize/2:

```
ssthresh = CongCtrlAlg()      // Target flight size in recovery
prr_delivered = 0             // Total bytes delivered in recovery
prr_out = 0                   // Total bytes sent in recovery
RecoverFS = snd.nxt - snd.una // FlightSize right before recovery
```

                           Figure 1

On every ACK starting or during the recovery:
```
DeliveredData = bytes newly cumulatively acknowledged
if (SACK is used) {
   DeliveredData += bytes newly selectively acknowledged
} else if (ACK is a DUPACK and prr_delivered < RecoverFS) {
   DeliveredData += MSS
}
if (DeliveredData is 0)
   Return


prr_delivered += DeliveredData
pipe = (RFC 6675 pipe algorithm)
safeACK = (snd.una advances and no further loss indicated)
if (pipe > ssthresh) {
   // Proportional Rate Reduction
   sndcnt = CEIL(prr_delivered * ssthresh / RecoverFS) - prr_out
} else {
   // PRR-CRB by default
   sndcnt = MAX(prr_delivered - prr_out, DeliveredData)
   if (safeACK) {
      // PRR-SSRB when recovery is in good progress
      sndcnt +=MSS
   }
   // Attempt to catch up, as permitted
   sndcnt = MIN(ssthresh - pipe, sndcnt)
}


if (prr_out is 0 AND sndcnt is 0) {
   // Force a fast retransmit upon entering recovery
   sndcnt = MSS
}
cwnd = pipe + sndcnt
```

                              Figure 2

   On any data transmission or retransmission:
      prr_out += (data sent)

                              Figure 3

7. Examples

   We illustrate these algorithms by showing their different behaviors
   for two scenarios: TCP experiencing either a single loss or a burst
   of 15 consecutive losses. In all cases we assume bulk data (no
   application pauses), standard Additive Increase Multiplicative
   Decrease (AIMD) congestion control [RFC5681], and cwnd = FlightSize =
   pipe = 20 segments, so ssthresh will be set to 10 at the beginning of
   recovery. We also assume standard Fast Retransmit and Limited
   Transmit [RFC3042], so TCP will send 2 new segments followed by 1
   retransmit in response to the first 3 duplicate ACKs following the
   losses.

   Each of the diagrams below shows the per ACK response to the first
   round trip for the various recovery algorithms when the zeroth
   segment is lost. The top line indicates the transmitted segment
   number triggering the ACKs, with an X for the lost segment. "cwnd"
   and "pipe" indicate the values of these algorithms after processing
   each returning ACK but before further (re)transmission. "Sent"
   indicates how much 'N'ew or 'R'etransmitted data would be sent. Note
   that the algorithms for deciding which data to send are out of scope
   of this document.

   RFC 6675
   ack#   X 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
   cwnd: 20 20 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
   pipe: 19 19 18 18 17 16 15 14 13 12 11 10 10 10 10 10 10 10 10
   sent: N N R   N N N N N N N N

   PRR
   ack#   X 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
   cwnd: 20 20 19 18 18 17 17 16 16 15 15 14 14 13 13 12 12 11 10
   pipe: 19 19 18 18 17 17 16 16 15 15 14 14 13 13 12 12 11 11 10
   sent: N N R   N   N   N   N   N   N   N   N

                              Figure 4

Note that both algorithms send the same total amount of data. RFC
6675 experiences a "half window of silence" while PRR spreads the
voluntary window reduction across an entire RTT.

Next, we consider the same initial conditions when the first 15
packets (0-14) are lost. During the remainder of the lossy RTT, only
5 ACKs are returned to the sender. We examine each of these
algorithms in succession.

```
RFC 6675
ack#   X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  15 16 17 18 19
cwnd:                                              20 20 11 11 11
pipe:                                              19 19  4 10 10
sent:                                               N  N 7R  R  R




PRR
ack#   X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  15 16 17 18 19
cwnd:                                              20 20  5  5  5
pipe:                                              19 19  4  4  4
sent:                                               N  N  R  R  R
```

                            Figure 5

In this specific situation, RFC 6675 is more aggressive because once
Fast Retransmit is triggered (on the ACK for segment 17), TCP
immediately retransmits sufficient data to bring pipe up to cwnd.
Our earlier measurements [RFC 6937 section 6] indicates that RFC 6675
significantly outperforms PRR, and some other similarly conservative
algorithms that we tested, showing that it is significantly common
for the actual losses to exceed the window reduction determined by
the congestion control algorithm.

Under such heavy losses, PRR uses the PRR-CRB to follow the packet
conservation principle. Since the total losses bring pipe below
ssthresh, data is sent such that the total data transmitted, prr_out,
follows the total data delivered to the receiver as reported by

returning ACKs. Transmission is controlled by the sending limit,
which is set to prr_delivered - prr_out. PRR-CRB's conservative
window reduction causes it to take excessively long to recover the
losses and exposes it to additional timeouts.

While not shown in the figure above, once the fast retransmits sent
upon ACK#17 deliver and solicit further ACKs that increment the
snd.una, PRR enters PRR-SSRB and increases the window by exactly 1
segment per ACK until pipe rises to ssthresh during recovery. On

heavy losses when cwnd is large, PRR-SSRB recovers the losses
exponentially faster than PRR-CRB. Although increasing the window
during recovery seems to be ill advised, it is important to remember
that this is actually less aggressive than permitted by RFC 5681,
which sends the same quantity of additional data as a single burst in
response to the ACK that triggered Fast Retransmit.

For less severe loss events, where the total losses are smaller than
the difference between FlightSize and ssthresh, PRR-CRB and PRR-SSRB
are not invoked since PRR stays in the proportional rate reduction
mode.

8. Properties

The following properties are common to both PRR-CRB and PRR-SSRB,
except as noted:

PRR maintains TCP's ACK clocking across most recovery events,
including burst losses. RFC 6675 can send large unclocked bursts
following burst losses.

Normally, PRR will spread voluntary window reductions out evenly
across a full RTT. This has the potential to generally reduce the
burstiness of Internet traffic, and could be considered to be a type
of soft pacing. Hypothetically, any pacing increases the probability
that different flows are interleaved, reducing the opportunity for
ACK compression and other phenomena that increase traffic burstiness.
However, these effects have not been quantified.

If there are minimal losses, PRR will converge to exactly the target
window chosen by the congestion control algorithm. Note that as TCP
approaches the end of recovery, prr_delivered will approach RecoverFS

and sndcnt will be computed such that prr_out approaches ssthresh.

Implicit window reductions, due to multiple isolated losses during
recovery, cause later voluntary reductions to be skipped. For small
numbers of losses, the window size ends at exactly the window chosen
by the congestion control algorithm.

For burst losses, earlier voluntary window reductions can be undone
by sending extra segments in response to ACKs arriving later during
recovery. Note that as long as some voluntary window reductions are
not undone, the final value for pipe will be the same as ssthresh,
the target cwnd value chosen by the congestion control algorithm.

PRR with either Reduction Bound improves the situation when there are
application stalls, e.g., when the sending application does not queue
data for transmission quickly enough or the receiver stops advancing

rwnd (receiver window). When there is an application stall early
during recovery, prr_out will fall behind the sum of the
transmissions permitted by sndcnt. The missed opportunities to send
due to stalls are treated like banked voluntary window reductions;
specifically, they cause prr_delivered - prr_out to be significantly
positive. If the application catches up while TCP is still in
recovery, TCP will send a partial window burst to catch up to exactly
where it would have been had the application never stalled. Although
this burst might be viewed as being hard on the network, this is
exactly what happens every time there is a partial RTT application
stall while not in recovery. We have made the partial RTT stall
behavior uniform in all states. Changing this behavior is out of
scope for this document.

PRR with Reduction Bound is less sensitive to errors in the pipe
estimator. While in recovery, pipe is intrinsically an estimator,
using incomplete information to estimate if un-SACKed segments are
actually lost or merely out of order in the network. Under some
conditions, pipe can have significant errors; for example, pipe is
underestimated when a burst of reordered data is prematurely assumed
to be lost and marked for retransmission. If the transmissions are
regulated directly by pipe as they are with RFC 6675, a step
discontinuity in the pipe estimator causes a burst of data, which
cannot be retracted once the pipe estimator is corrected a few ACKs
later. For PRR, pipe merely determines which algorithm, PRR or the

Reduction Bound, is used to compute sndcnt from DeliveredData. While
pipe is underestimated, the algorithms are different by at most 1
segment per ACK. Once pipe is updated, they converge to the same
final window at the end of recovery.

Under all conditions and sequences of events during recovery, PRR-CRB
strictly bounds the data transmitted to be equal to or less than the
amount of data delivered to the receiver. We claim that this Strong
Packet Conservation Bound is the most aggressive algorithm that does
not lead to additional forced losses in some environments. It has
the property that if there is a standing queue at a bottleneck with
no cross traffic, the queue will maintain exactly constant length for
the duration of the recovery, except for +1/-1 fluctuation due to
differences in packet arrival and exit times. See Appendix A for a
detailed discussion of this property.

Although the Strong Packet Conservation Bound is very appealing for a
number of reasons, our earlier measurements [RFC 6937 section 6]
demonstrate that it is less aggressive and does not perform as well
as RFC 6675, which permits bursts of data when there are bursts of
losses. PRR-SSRB is a compromise that permits TCP to send 1 extra
segment per ACK as compared to the Packet Conserving Bound when the
ACK indicates the recovery is in good progress without further

losses. From the perspective of a strict Packet Conserving Bound,
PRR-SSRB does indeed open the window during recovery; however, it is
significantly less aggressive than RFC 6675 in the presence of burst
losses.

9. Adapting PRR to other transport protocols

The main PRR algorithm and reductions bounds can be adapted to any
transport that can support RFC 6675. In one major implementation
(Linux TCP), PRR has been the default fast recovery algorithm for its
default and supported congestion control modules.

The safeACK heuristic can be generalized as any ACK of a
retransmission that does not cause some other segment to be marked
for retransmission. That is, PRR_SSRB is safe on any ACK that
reduces the total number of pending and outstanding retransmissions.

10. Acknowledgements

This document is based in part on previous incomplete work by Matt
Mathis, Jeff Semke, and Jamshid Mahdavi [RHID] and influenced by
several discussions with John Heffner.

Monia Ghobadi and Sivasankar Radhakrishnan helped analyze the
experiments.

Ilpo Jarvinen reviewed the initial implementation.

Neal Cardwell and Mark Allman improved the document through their
insightful reviews.

11. Security Considerations

PRR does not change the risk profile for TCP.

Implementers that change PRR from counting bytes to segments have to
be cautious about the effects of ACK splitting attacks [Savage99],
where the receiver acknowledges partial segments for the purpose of
confusing the sender's congestion accounting.

12. Normative References

   [RFC0793] Postel, J., "Transmission Control Protocol", STD 7,
              RFC 793, DOI 10.17487/RFC0793, September 1981,
              <https://www.rfc-editor.org/info/rfc793>.

   [RFC2018] Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, "TCP
              Selective Acknowledgment Options", RFC 2018,
              DOI 10.17487/RFC2018, October 1996,
              <https://www.rfc-editor.org/info/rfc2018>.

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119,
              DOI 10.17487/RFC2119, March 1997,
              <https://www.rfc-editor.org/info/rfc2119>.

   [RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion

              Control", RFC 5681, DOI 10.17487/RFC5681, September 2009,
              <https://www.rfc-editor.org/info/rfc5681>.

   [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, DOI 10.17487/RFC6675, August 2012,
              <https://www.rfc-editor.org/info/rfc6675>.

   [RFC8985] Cheng, Y., Cardwell, N., Dukkipati, N., and P. Jha, "The
              RACK-TLP Loss Detection Algorithm for TCP", RFC 8985,
              DOI 10.17487/RFC8985, February 2021,
              <https://www.rfc-editor.org/info/rfc8985>.

13. Informative References

   [CUBIC]   Rhee, I. and L. Xu, "CUBIC: A new TCP-friendly high-speed
              TCP variant", PFLDnet 2005, February 2005.

   [FACK]   Mathis, M. and J. Mahdavi, "Forward Acknowledgment:
              Refining TCP Congestion Control", ACM SIGCOMM SIGCOMM96,
              August 1996.

   [Flach2016policing]
              Flach, T., Papageorge, P., Terzis, A., Pedrosa, L., Cheng,
              Y., Al Karim, T., Katz-Bassett, E., and R. Govindan, "An
              Internet-Wide Analysis of Traffic Policing", ACM
              SIGCOMM SIGCOMM2016, August 2016.

   [IMC11]   Dukkipati, N., Mathis, M., Cheng, Y., and M. Ghobadi,
              "Proportional Rate Reduction for TCP", Proceedings of the
              11th ACM SIGCOMM Conference on Internet Measurement
              2011, Berlin, Germany, November 2011.

   [Jacobson88]
              Jacobson, V., "Congestion Avoidance and Control", SIGCOMM
              Comput. Commun. Rev. 18(4), August 1988.

   [Laminar] Mathis, M., "Laminar TCP and the case for refactoring TCP
              congestion control", Work in Progress, 16 July 2012.

   [RFC3042] Allman, M., Balakrishnan, H., and S. Floyd, "Enhancing

               TCP's Loss Recovery Using Limited Transmit", [RFC 3042](https://www.rfc-editor.org/info/rfc3042),
               DOI 10.17487/RFC3042, January 2001,
               <https://www.rfc-editor.org/info/rfc3042>.

   [RFC3168]   Ramakrishnan, K., Floyd, S., and D. Black, "The Addition
               of Explicit Congestion Notification (ECN) to IP",
               [RFC 3168](https://www.rfc-editor.org/info/rfc3168), DOI 10.17487/RFC3168, September 2001,
               <https://www.rfc-editor.org/info/rfc3168>.

   [RFC3517]   Blanton, E., Allman, M., Fall, K., and L. Wang, "A
               Conservative Selective Acknowledgment (SACK)-based Loss
               Recovery Algorithm for TCP", [RFC 3517](https://www.rfc-editor.org/info/rfc3517),
               DOI 10.17487/RFC3517, April 2003,
               <https://www.rfc-editor.org/info/rfc3517>.

   [RFC6937]   Mathis, M., Dukkipati, N., and Y. Cheng, "Proportional
               Rate Reduction for TCP", [RFC 6937](https://www.rfc-editor.org/info/rfc6937), DOI 10.17487/RFC6937,
               May 2013, <https://www.rfc-editor.org/info/rfc6937>.

   [RHID]   Mathis, M., Semke, J., and J. Mahdavi, "The Rate-Halving
               Algorithm for TCP Congestion Control", Work in Progress,
               August 1999.

   [RHweb]   Mathis, M. and J. Mahdavi, "TCP Rate-Halving with Bounding
               Parameters", Web publication, December 1997,
               <http://www.psc.edu/networking/papers/FACKnotes/current/>.

   [Savage99] Savage, S., Cardwell, N., Wetherall, D., and T. Anderson,
               "TCP congestion control with a misbehaving receiver",
               SIGCOMM Comput. Commun. Rev. 29(5), October 1999.

[Appendix A](#). Strong Packet Conservation Bound

   PRR-CRB is based on a conservative, philosophically pure, and
   aesthetically appealing Strong Packet Conservation Bound, described
   here. Although inspired by the packet conservation principle
   [Jacobson88], it differs in how it treats segments that are missing
   and presumed lost. Under all conditions and sequences of events
   during recovery, PRR-CRB strictly bounds the data transmitted to be
   equal to or less than the amount of data delivered to the receiver.
   Note that the effects of presumed losses are included in the pipe
   calculation, but do not affect the outcome of PRR-CRB, once pipe has
   fallen below ssthresh.

We claim that this Strong Packet Conservation Bound is the most aggressive algorithm that does not lead to additional forced losses in some environments. It has the property that if there is a standing queue at a bottleneck that is carrying no other traffic, the queue will maintain exactly constant length for the entire duration of the recovery, except for +1/-1 fluctuation due to differences in packet arrival and exit times. Any less aggressive algorithm will result in a declining queue at the bottleneck. Any more aggressive algorithm will result in an increasing queue or additional losses if it is a full drop tail queue.

We demonstrate this property with a little thought experiment:

Imagine a network path that has insignificant delays in both directions, except for the processing time and queue at a single bottleneck in the forward path. By insignificant delay, we mean when a packet is "served" at the head of the bottleneck queue, the following events happen in much less than one bottleneck packet time: the packet arrives at the receiver; the receiver sends an ACK that arrives at the sender; the sender processes the ACK and sends some data; the data is queued at the bottleneck.

If sndcnt is set to DeliveredData and nothing else is inhibiting sending data, then clearly the data arriving at the bottleneck queue will exactly replace the data that was served at the head of the queue, so the queue will have a constant length. If queue is drop tail and full, then the queue will stay exactly full. Losses or reordering on the ACK path only cause wider fluctuations in the queue size, but do not raise its peak size, independent of whether the data is in order or out of order (including loss recovery from an earlier RTT). Any more aggressive algorithm that sends additional data will overflow the drop tail queue and cause loss. Any less aggressive algorithm will under-fill the queue. Therefore, setting sndcnt to DeliveredData is the most aggressive algorithm that does not cause forced losses in this simple network. Relaxing the assumptions (e.g., making delays more authentic and adding more flows, delayed ACKs, etc.) is likely to increase the fine grained fluctuations in queue size but does not change its basic behavior.

Note that the congestion control algorithm implements a broader notion of optimal that includes appropriately sharing the network. Typical congestion control algorithms are likely to reduce the data sent relative to the Packet Conserving Bound implemented by PRR, bringing TCP's actual window down to ssthresh.

Authors' Addresses

Matt Mathis
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, California 94043
United States of America
Email: mattmathis@google.com

Nandita Dukkipati
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, California 94043
United States of America
Email: nanditad@google.com

Yuchung Cheng
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, California 94043
United States of America
Email: ycheng@google.com