TCP Maintenance Working Group
Internet-Draft
Intended status: Experimental
Expires: August 27, 2012

M. Mathis
N. Dukkipati
Y. Cheng
Google, Inc

February 24, 2012

Proportional Rate Reduction for TCP
draft-ietf-tcpm-proportional-rate-reduction-01.txt

Abstract

This document describes an experimental algorithm, Proportional Rate Reduction (PPR) to improve the accuracy of the amount of data sent by TCP during loss recovery. Standard Congestion Control requires that TCP and other protocols reduce their congestion window in response to losses. This window reduction naturally occurs in the same round trip as the data retransmissions to repair the losses, and is implemented by choosing not to transmit any data in response to some ACKs arriving from the receiver. Two widely deployed algorithms are used to implement this window reduction: Fast Recovery and Rate Halving. Both algorithms are needlessly fragile under a number of conditions, particularly when there is a burst of losses that such that the number of ACKs returning to the sender is small. Proportional Rate Reduction minimizes these excess window reductions such that at the end of recovery the actual window size will be as close as possible to ssthresh, the window size determined by the congestion control algorithm. It is patterned after Rate Halving, but using the fraction that is appropriate for target window chosen by the congestion control algorithm.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction 3
2. Definitions 4
3. Algorithms 5
  3.1. Examples 6
4. Properties 9
5. Measurements 11
6. Conclusion and Recommendations 12
7. Acknowledgements 13
8. Security Considerations 13
9. IANA Considerations 13
10. References 13
Appendix A. Packet Conservation Bound 14
Authors' Addresses 15

1. Introduction

This document describes an experimental algorithm, Proportional Rate Reduction (PPR) to improve the accuracy of the amount of data sent by TCP during loss recovery.

Standard Congestion Control [RFC5681] requires that TCP (and other protocols) reduce their congestion window in response to losses. Fast Recovery, described in the same document, is the reference algorithm for making this adjustment. It's 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 adjusts the window by waiting for one half RTT of ACKs to pass before sending any data. It is fragile because it can not compensate for the implicit window reduction caused by the losses themselves, and is exposed to timeouts. For example if half of the data or ACKs are lost, Fast Recovery's expected behavior would be wait for half window of ACKs to pass and then not receive any ACKs for the recovery and suffer a timeout.

The rate-halving algorithm improves this situation by sending data on alternate ACKs during recovery, such that after one RTT the window has been halved. Rate-having is implemented in Linux after only being informally published [RHweb], including an uncompleted Internet-Draft [RHID]. Rate-halving does not adequately compensate for the implicit window reduction caused by the losses and assumes a 50% window reduction, which was completely standard at the time it was written, but not appropriate for modern congestion control algorithms such as Cubic [CUBIC], which can reduce the window by less than 50%. As a consequence rate-halving often allows the window to fall further than necessary, reducing performance and increasing the risk of timeouts if there are additional losses.

Proportional Rate Reduction (PPR) avoids these excess window reductions such that at the end of recovery the actual window size will be as close as possible to, ssthresh, the window size determined by the congestion control algorithm. It is patterned after Rate Halving, but using the fraction that is appropriate for 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 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 one segment per ACK. PRR-CRB meets the strong conservative bound described in Appendix A, however in

real networks it does not perform as well as the algorithms described in RFC 3517, which prove to be non-conservative in a significant number of cases. SSRB offers a compromise by allowing TCP to send one additional segment per ACK relative to CRB in some situations. Although SSRB is less aggressive than RFC 3517 (transmitting fewer segments or taking more time to transmit them) it outperforms it, due to the lower probability of additional losses during recovery.

PRR and both reduction bounds are based on common design principles, derived from 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 Proportional Rate Reduction and the reduction bound rely on this self clock process, and are only slightly affected by the accuracy of other estimators, such as pipe [RFC3517] and cwnd. This is what gives the algorithms their precision in the presence of events that cause uncertainty in other estimators.

We evaluated these and other algorithms in a large scale measurement study, summarized below. The most important results from that study are presented in an companion paper [IMC11]. PRR+SSRB outperforms both RFC 3517 and Linux Rate Halving under authentic network traffic, even though it is less aggressive than RFC 3517.

The algorithms are described as modifications to RFC 5681 [RFC5681], TCP Congestion Control, using concepts drawn from the pipe algorithm [RFC3517]. They are most accurate and more easily implemented with SACK [RFC2018], but they do not require SACK.

2. Definitions

The following terms, parameters and state variables are used as they are defined in earlier documents:

RFC 3517: covered (as in "covered sequence numbers")

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

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.

We define some additional variables:

SACKd: The total number of bytes that the scoreboard indicates have been delivered to the receiver. This can be computed by scanning the

scoreboard and counting the total number of bytes covered by all sack blocks. If SACK is not in use, SACKd is not defined.

DeliveredData: The total number of bytes that the current ACK indicates have been delivered to the receiver. When not in recovery, DeliveredData is the change in snd.una. With SACK, DeliveredData can be computed precisely as the change in snd.una plus the (signed) change in SACKd. In recovery without SACK, DeliveredData is estimated to be 1 SMSS on duplicate acknowledgements, and on a subsequent partial or full ACK, DeliveredData is estimated to be the change in snd.una, minus one SMSS for each preceding duplicate ACK.

Note that DeliveredData is robust: for TCP using SACK, DeliveredData can be precisely computed anywhere in the network just by inspecting the returning ACKs. The consequence of missing ACKs is that later ACKs will show a larger DeliveredData. Furthermore, for any TCP (with or without SACK) the sum of DeliveredData must agree with the forward progress over the same time interval.

We introduce a local variable "sndcnt", which indicates 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.

3. Algorithms

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

```
   ssthresh = CongCtrlAlg()  // Target cwnd after recovery
   prr_delivered = 0         // Total bytes delivered during recov
   prr_out = 0               // Total bytes sent during recovery
   RecoverFS = snd.nxt-snd.una // FlightSize at the start of recov
```

On every ACK during recovery compute:

```
   DeliveredData = delta(snd.una) + delta(SACKd)
   prr_delivered += DeliveredData
   pipe = (RFC 3517 pipe algorithm)
   if (pipe > ssthresh) {
      // Proportional Rate Reduction
      sndcnt = CEIL(prr_delivered * ssthresh / RecoverFS) - prr_out
   } else {
      // Two version of the reduction bound
      if (conservative) {    // PRR+CRB
        limit = prr_delivered - prr_out
      } else {               // PRR+SSRB
        limit = MAX(prr_delivered - prr_out, DeliveredData) + MSS
      }
      // Attempt to catch up, as permitted by limit
      sndcnt = MIN(ssthresh - pipe, limit)
   }
```

On any data transmission or retransmission:

```
      prr_out += (data sent) // strictly less than or equal to sndcnt
```

3.1. 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, standard AIMD congestion control 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, so we send two new segments followed by one retransmit on the first 3 duplicate ACKs after 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. "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.

When there is a single loss, PRR with either of the reduction bound algorithms has the same behavior. We show "RB", a flag indicating which reduction bound subexpression ultimately determined the value of sndcnt. When there is minimal losses "limit" (both algorithms) will always be larger than ssthresh - pipe, so the sndcnt will be ssthresh - pipe indicated by "s" in the "RB" row. Since PRR does not use cwnd during recovery it is not shown in the example.

```
   RFC 3517
   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
```

Rate Halving (Linux)

```
   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 11
   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


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

Note that all three algorithms send same total amount of data. RFC 3517 experiences a "half-window of silence", while the Rate Halving and PRR spread 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 3517

```
   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


   Rate Halving (Linux)
   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


   PRR-CRB
   ack#   X  X  X  X  X  X  X  X  X  X  X  X  X  X  X 15 16 17 18 19
   pipe:                                              19 19  4  4  4
   sent:                                               N  N  R  R  R
   RB:                                                       f  f  f


   PRR-SSRB
   ack#   X  X  X  X  X  X  X  X  X  X  X  X  X  X  X 15 16 17 18 19
   pipe:                                              19 19  4  5  6
   sent:                                               N  N 2R 2R 2R
   RB:                                                       d  d  d
```

In this specific situation, RFC 3517 is very non-conservative, because once fast retransmit is triggered (on the ACK for segment 17) TCP immediately retransmits sufficient data to bring pipe up to cwnd. Our measurement data (see Section 5) indicates that RFC 3517 significantly outperforms Rate Halving, PRR-CRB and some other similarly conservative algorithms that we tested, suggesting that it is significantly common for the actual losses to exceed the window reduction determined by the congestion control algorithm.

The Linux implementation of Rate Halving includes an early version of the conservative reduction bound [RHweb]. In this situation the five ACKs trigger exactly one transmission each (2 new data, 3 old data), and cwnd is set to 5. At a window size of 5, it takes three round trips to retransmit all 15 lost segments. Rate Halving does not raise the window at all during recovery, so when recovery finally completes, TCP will slowstart cwnd from 5 up to 10. In this example, TCP operates at half of the window chosen by the congestion control for more than three RTTs, increasing the elapsed time and exposing it to timeouts in the event that there are additional losses.

PRR-CRB implements conservative reduction bound. 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 was set to prr_delivered - prr_out. This is indicated by the RB:f tagging in the figure. In this case PRR-CRB is exposed to exactly the same problems as Rate Halving, the excess window reduction causes it to take excessively long to recover the losses and exposes it to additional timeouts.

PRR-SSRB increases the window by exactly 1 segment per ACK until pipe rises to sshthresh during recovery. This is accomplished by setting limit to one greater than the data reported to have been delivered to the receiver on this ACK, implementing slowstart during recovery, and indicated by RB:d tagging in the figure. Although increasing the window during recovery seems to be ill advised, it is important to remember that this 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 extreme events, where the total losses are smaller than the difference between Flight Size and ssthresh, PRR-CRB and PRR-SSRB have identical behaviours.

4. Properties

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

Proportional Rate Reduction maintains TCPs ACK clocking across most recovery events, including burst losses. RFC 3517 can send large unclocked bursts following burst losses.

Normally Proportional Rate Reduction 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, Proportional Rate Reduction 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.

Proportional Rate Reduction 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). 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.

Proportional Rate Reduction with Reduction Bound is significantly less sensitive to errors of the pipe estimator. While in recovery, pipe is intrinsically an estimator, using incomplete information to guess if un-SACKed segments are actually lost or out-of-order in the network. Under some conditions pipe can have significant errors, for example when a burst of reordered data is presumed to be lost and is retransmitted, but then the original data arrives before the retransmission. If the transmissions are regulated directly by pipe as they are in RFC 3517, then errors and discontinuities in the value of the pipe estimator can cause significant errors in the amount of data sent. With Proportional Rate Reduction with Reduction Bound, pipe merely determines how sndcnt is computed from DeliveredData. Since short term errors in pipe are smoothed out across multiple ACKs and both Proportional Rate Reduction and the reduction bound converge to the same final window, errors in the pipe estimator have less impact on the final outcome.

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 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 packet Packet Conserving Bound in very appealing for a number of reasons, our measurements summarized in Section 5 demonstrate that it is less aggressive and does not perform as well as RFC3517, which permits large bursts of data when there are bursts of losses. PRR-SSRB is a compromise that permits TCP to send one extra segment per ACK as compared to the packet conserving bound. From the perspective of the packet conserving bound, PRR-SSRB does indeed open the window during recovery, however it is significantly less aggressive than RFC3517 in the presence of burst losses.

5. Measurements

In a companion IMC11 paper [IMC11] we describe some measurements comparing the various strategies for reducing the window during recovery. The results are summarized here.

The various window reduction algorithms and extensive instrumentation were all implemented in Linux 2.6. We used the uniform set of algorithms present in the base Linux implementation, including CUBIC [CUBIC], limited transmit [RFC3742], threshold transmit from [FACK] and lost retransmission detection algorithms. We confirmed that the behaviors of Rate Halving (the Linux default), RFC 3517 and PRR were authentic to their respective specifications and that performance and features were comparable to the kernels in production use. The different window reduction algorithms were all present in the same kernel and could be selected with a sysctl, such that we had an absolutely uniform baseline for comparing them.

Our experiments included an additional algorithm, PRR with an unlimited bound (PRR-UB), which sends ssthresh-pipe bursts when pipe falls below ssthresh. This behavior parallels RFC 3517.

An important detail of this configuration is that CUBIC only reduces the window by 30%, as opposed to the 50% reduction used by traditional congestion control algorithms. This, in conjunction with using only standard algorithms to trigger Fast Retransmit, accentuates the tendency for RFC 3517 and PRR-UB to send a burst at the point when Fast Retransmit gets triggered if pipe is already below ssthresh.

All experiments were performed on servers carrying production traffic for multiple Google services.

In this configuration it is observed that for 32% of the recovery events, pipe falls below ssthresh before Fast Retransmit is triggered, thus the various PRR algorithms start in the reduction bound phase, and RFC 3517 send bursts of segments with the fast retransmit.

In the companion paper we observe that PRR-SSRB spends the least time in recovery of all the algorithms tested, largely because it experiences fewer timeouts once it is already in recovery.

RFC 3517 experiences 29% more detected lost retransmissions and 2.6% more timeouts (presumably due to undetected lost retransmissions) than PRR-SSRB. These results are representative of PRR-UB and other algorithms that send bursts when pipe falls below ssthresh.

Rate Halving experiences 5% more timeouts and significantly smaller final cwnd values at the end of recovery. The smaller cwnd sometimes causes the recovery itself to take extra round trips. These results are representative of PRR-CRB and other algorithms that implement strict packet conservation during recovery.

6. Conclusion and Recommendations

Although the packet conserving bound is very appealing for a number of reasons, our measurements demonstrate that it is less aggressive and does not perform as well as RFC3517, which permits significant bursts of data when there are large bursts of losses. PRR-SSRB is a compromise that permits TCP to send one extra segment per ACK as relative to the packet conserving bound. From the perspective of the packet conserving bound, PRR-SSRB does indeed open the window during recovery, however it is significantly less aggressive than RFC3517 in the presence of burst losses. Even so, it often out performs RFC3517, because it avoids some of the self inflicted losses caused by bursts from RFC3517.

At this time we see no reason not to test and deploy PRR-SSRB on a large scale. Implementers worried about any potential impact of raising the window during recovery may want to optionally support PRR-CRB (which is actually simpler to implement) for comparison studies.

One final comment about terminology: we expect that common usage will drop "slow start reduction bound" from the algorithm name. This document needed to be pedantic about having distinct names for

proportional rate reduction and every variant of the reduction bound. However, once paired they become one.

7. Acknowledgements

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

Monia Ghobadi and Sivasankar Radhakrishnan helped analyze the experiments.

Ilpo Jarvinen for reviewing the code.

8. Security Considerations

Proportional Rate Reduction 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.

9. IANA Considerations

This document makes no request of IANA.

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

10. References

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

[RFC3517] Blanton, E., Allman, M., Fall, K., and L. Wang, "A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP", RFC 3517, April 2003.

[RFC3742] Floyd, S., "Limited Slow-Start for TCP with Large Congestion Windows", RFC 3742, March 2004.

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

[IMC11] Dukkipati, N., Mathis, M., and Y. Cheng, "Proportional Rate Reduction for TCP", ACM Internet Measurement Conference IMC11, December 2011.

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

[RHID] Mathis, M., Semke, J., Mahdavi, J., and K. Lahey, "The Rate-Halving Algorithm for TCP Congestion Control", draft-ratehalving (work in progress), June 1999.

[RHweb]   Mathis, M. and J. Mahdavi, "TCP Rate-Halving with Bounding Parameters", Web publication , December 1997.

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

[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. Packet Conservation Bound

PRR-CRB meets a conservative, philosophically pure and aesthetically appealing notion of correct, described here. However, in real networks it does not perform as well as the algorithms described in RFC 3517, which proves to be non-conservative in a significant number of cases.

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 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, I 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; which 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 the 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 which sends additional data will cause a queue overflow and 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) may increases 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 of 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 Amphitheater Parkway
Mountain View, California 93117
USA

Email: mattmathis@google.com

Nandita Dukkipati
Google, Inc
1600 Amphitheater Parkway
Mountain View, California 93117
USA

Email: nanditad@google.com

Yuchung Cheng
Google, Inc
1600 Amphitheater Parkway
Mountain View, California 93117
USA

Email: ycheng@google.com