TCP Maintenance Working Group
Internet-Draft
Intended status: Experimental
Expires: August 29, 2013

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

February 25, 2013

Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses

draft-dukkipati-tcpm-tcp-loss-probe-01.txt

Abstract

Retransmission timeouts are detrimental to application latency, especially for short transfers such as Web transactions where timeouts can often take longer than all of the rest of a transaction. The primary cause of retransmission timeouts are lost segments at the tail of transactions. This document describes an experimental algorithm for TCP to quickly recover lost segments at the end of transactions or when an entire window of data or acknowledgments are lost. Tail Loss Probe (TLP) is a sender-only algorithm that allows the transport to recover tail losses through fast recovery as opposed to lengthy retransmission timeouts. If a connection is not receiving any acknowledgments for a certain period of time, TLP transmits the last unacknowledged segment (loss probe). In the event of a tail loss in the original transmissions, the acknowledgment from the loss probe triggers SACK/FACK based fast recovery. TLP effectively avoids long timeouts and thereby improves TCP performance.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

Retransmission timeouts are detrimental to application latency, especially for short transfers such as Web transactions where timeouts can often take longer than all of the rest of a transaction. This document describes an experimental algorithm, Tail Loss Probe (TLP), to invoke fast recovery for losses that would otherwise be only recoverable through timeouts.

The Transmission Control Protocol (TCP) has two methods for recovering lost segments. First, the fast retransmit algorithm relies on incoming duplicate acknowledgments (ACKs), which indicate that the receiver is missing some data. After a required number of duplicate ACKs have arrived at the sender, it retransmits the first unacknowledged segment and continues with a loss recovery algorithm such as the SACK-based loss recovery [RFC6675]. If the fast retransmit algorithm fails for any reason, TCP uses a retransmission timeout as the last resort mechanism to recover lost segments. If an ACK for a given segment is not received in a certain amount of time called retransmission timeout (RTO), the segment is resent [RFC6298].

Timeouts can occur in a number of situations, such as the following:

(1) Drop tail at the end of transactions. Example: consider a transfer of five segments sent on a connection that has a congestion window of ten. Any degree of loss in the tail, such as segments four and five, will only be recovered via a timeout.

(2) Mid-transaction loss of an entire window of data or ACKs. Unlike (1) there is more data waiting to be sent. Example: consider a transfer of four segments to be sent on a connection that has a congestion window of two. If the sender transmits two segments and both are lost then the loss will only be recovered via a timeout.

(3) Insufficient number of duplicate ACKs to trigger fast recovery at sender. The early retransmit mechanism [RFC5827] addresses this problem in certain special circumstances, by reducing the number of duplicate ACKs required to trigger a fast retransmission.

(4) An unexpectedly long round-trip time (RTT), such that the ACKs arrive after the RTO timer expires. The F-RTO algorithm [RFC5682] is designed to detect such spurious retransmission timeouts and at least partially undo the consequences of such events.

Measurements on Google Web servers show that approximately 70% of retransmissions for Web transfers are sent after the RTO timer expires, while only 30% are handled by fast recovery. Even on servers exclusively serving YouTube videos, RTO based retransmissions account for about 46% of the retransmissions. If the losses are detectable from the ACK stream (through duplicate ACKs or SACK blocks) then early retransmit, fast recovery and proportional rate reduction are effective in avoiding timeouts [IMC11PRR]. Timeout retransmissions that occur in recovery and disorder state (a state indicating that a connection has received some duplicate ACKs), account for just 4% of the timeout episodes. On the other hand 96% of the timeout episodes occur without any preceding duplicate ACKs or other indication of losses at the sender [IMC11PRR]. Early retransmit and fast recovery have no hope of repairing losses without these indications. Efficiently addressing situations that would cause timeouts without any prior indication of losses is a significant opportunity for additional improvements to loss recovery.

To get a sense of just how long the RTOs are in relation to connection RTTs, following is the distribution of RTO/RTT values on Google Web servers. [percentile, RTO/RTT]: [50th percentile, 4.3]; [75th percentile, 11.3]; [90th percentile, 28.9]; [95th percentile, 53.9]; [99th percentile, 214]. Large RTOs, typically caused by variance in measured RTTs, can be a result of intermediate queuing, and service variability in mobile channels. Such large RTOs make a huge contribution to the long tail on the latency statistics of short flows. Note that simply reducing the length of RTO does not address the latency problem for two reasons: first, it increases the chances of spurious retransmissions. Second and more importantly, an RTO reduces TCP's congestion window to one and forces a slow start. Recovery of losses without relying primarily on the RTO mechanism is beneficial for short TCP transfers.

The question we address in this document is: Can a TCP sender recover tail losses of transactions through fast recovery and thereby avoid lengthy retransmission timeouts? We specify an algorithm, Tail Loss Probe (TLP), which sends probe segments to trigger duplicate ACKs with the intent of invoking fast recovery more quickly than an RTO at the end of a transaction. TLP is applicable only for connections in Open state, wherein a sender is receiving in-sequence ACKs and has not detected any lost segments. TLP can be implemented by modifying only the TCP sender, and does not require any TCP options or changes to the receiver for its operation. For convenience, this document mostly refers to TCP, but the algorithms and other discussion are valid for Stream Control Transmission Protocol (SCTP) as well.

This document is organized as follows. Section 2 describes the basic Loss Probe algorithm. Section 3 outlines an algorithm to detect the cases when TLP plugs a hole in the sender. The algorithm makes the sender aware that a loss had occurred so it performs the appropriate congestion window reduction. Section 4 discusses the interaction of TLP with early retransmit in being able to recover any degree of tail losses. Section 5 discusses the experimental results with TLP on Google Web servers. Section 6 discusses related work, and Section 7 discusses the security considerations.

1.1. Terminology

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].

2. Loss probe algorithm

The Loss probe algorithm is designed for a sender to quickly detect tail losses without waiting for an RTO. We will henceforth use tail loss to generally refer to either drops at the tail end of transactions or a loss of an entire window of data/ACKs. TLP works for senders with SACK enabled and in Open state, i.e. the sender has so far received in-sequence ACKs with no SACK blocks. The risk of a sender incurring a timeout is high when the sender has not received any ACKs for a certain portion of time but is unable to transmit any further data either because it is application limited (out of new data to send), receiver window (rwnd) limited, or congestion window (cwnd) limited. For these circumstances, the basic idea of TLP is to transmit probe segments for the specific purpose of eliciting additional ACKs from the receiver. The initial idea was to send some form of zero window probe (ZWP) with one byte of new or old data. The ACK from the ZWP would provide an additional opportunity for a SACK block to detect loss without an RTO. Additional losses can be detected subsequently and repaired as SACK based fast recovery proceeds. However, in practice sending a single byte of data turned out to be problematic to implement and more fragile than necessary. Instead we use a full segment to probe but have to add complexity to compensate for the probe itself masking losses.

Define probe timeout (PTO) to be a timer event indicating that an ACK is overdue on a connection. The PTO value is set to max(2 * SRTT, 10ms), where SRTT is the smoothed round-trip time [RFC6298], and is adjusted to account for delayed ACK timer when there is only one outstanding segment.

The basic version of the TLP algorithm transmits one probe segment after a probe timeout if the connection has outstanding unacknowledged data but is otherwise idle, i.e. not receiving any ACKs or is cwnd/rwnd/application limited. The transmitted segment, aka loss probe, can be either a new segment if available and the receive window permits, or a retransmission of the most recently sent segment, i.e., the segment with the highest sequence number. When there is tail loss, the ACK from the probe triggers fast recovery. In the absence of loss, there is no change in the congestion control or loss recovery state of the connection, apart from any state related to TLP itself.

TLP MUST NOT be used for non-SACK connections. SACK feedback allows senders to use the algorithm described in section 3 to infer whether any segments were lost.

2.1. Pseudocode

We define the terminology used in specifying the TLP algorithm:

FlightSize: amount of outstanding data in the network as defined in [RFC5681].

RTO: The transport's retransmission timeout (RTO) is based on measured round-trip times (RTT) between the sender and receiver, as specified in [RFC6298] for TCP.

PTO: Probe timeout is a timer event indicating that an ACK is overdue. Its value is constrained to be smaller than or equal to an RTO.

SRTT: smoothed round-trip time computed like in [RFC6298].

Open state: the sender has so far received in-sequence ACKs with no SACK blocks, and no other indications (such as retransmission timeout) that a loss may have occurred.

Consecutive PTOs: back-to-back PTOs all scheduled for the same tail packets in a flight. The (N+1)st PTO is scheduled after transmitting the probe segment for Nth PTO.

The TLP algorithm works as follows:

(1) Schedule PTO after transmission of new data in Open state:

```
 Check for conditions to schedule PTO outlined in step 2 below.
 FlightSize > 1: schedule PTO in max(2*SRTT, 10ms).
 FlightSize == 1: schedule PTO in max(2*SRTT, 1.5*SRTT+WCDelAckT).
 If RTO is earlier, schedule PTO in its place: PTO = min(RTO, PTO).
```

WCDelAckT stands for worst case delayed ACK timer. When FlightSize is 1, PTO is inflated additionally by WCDelAckT time to compensate for a potential long delayed ACK timer at the receiver. The RECOMMENDED value for WCDelAckT is 200ms.

A PTO value of 2*SRTT allows a sender to wait long enough to know that an ACK is overdue. Under normal circumstances, i.e. no losses, an ACK typically arrives in one RTT. But choosing PTO to be exactly an RTT is likely to generate spurious probes given that even end-system timings can easily push an ACK to be above an RTT. We chose PTO to be the next integral value of RTT. If RTO is smaller than the computed value for PTO, then a probe is scheduled to be sent at the RTO time. The RTO timer is rearmed at the time of sending the probe, as is shown in Step 3 below. This ensures that a PTO is always sent prior to a connection experiencing an RTO.

(2) Conditions for scheduling PTO:

(a) Connection is in Open state.
(b) Connection is either cwnd limited or application limited.
(c) Number of consecutive PTOs <= 2.
(d) Connection is SACK enabled.

Implementations MAY use one or two consecutive PTOs.

(3) When PTO fires:

```
(a) If a new previously unsent segment exists:
      -> Transmit new segment.
      -> FlightSize += SMSS. cwnd remains unchanged.
(b) If no new segment exists:
      -> Retransmit the last segment.
(c) Increment statistics counter for loss probes.
(d) If conditions in (2) are satisfied:
      -> Reschedule next PTO.
    Else:
      -> Rearm RTO to fire at epoch 'now+RTO'.
```

The reason for retransmitting the last segment in Step (b) is so that the ACK will carry SACK blocks and trigger either SACK-based loss recovery [RFC6675] or FACK threshold based fast recovery [FACK]. On transmission of a TLP, a MIB counter is incremented to keep track of the total number of loss probes sent.

(4) During ACK processing:

```
Cancel any existing PTO.
If conditions in (2) allow:
  -> Reschedule PTO relative to the ACK receipt time.
```

Following is an example of TLP. All events listed are at a TCP sender.

(1) Sender transmits segments 1-10: 1, 2, 3, ..., 8, 9, 10. There is no more new data to transmit. A PTO is scheduled to fire in 2 RTTs, after the transmission of the 10th segment.

(2) Receives acknowledgements (ACKs) for segments 1-5; segments 6-10 are lost and no ACKs are received. Note that the sender (re)schedules its PTO timer relative to the last received ACK, which is the ACK for segment 5 in this case. The sender sets the PTO interval using the calculation described in step (1) of the algorithm.

(3) When PTO fires, sender retransmits segment 10.

(4) After an RTT, SACK for packet 10 arrives. The ACK also carries SACK holes for segments 6, 7, 8 and 9. This triggers FACK threshold based recovery.

(5) Connection enters fast recovery and retransmits remaining lost segments.

2.2. FACK threshold based recovery

At the core of TLP is its reliance on FACK threshold based algorithm to invoke Fast Recovery. In this section we specify this algorithm.

Section 3.1 of the Forward Acknowledgement (FACK) Paper [FACK] describes an alternate algorithm for triggering fast retransmit, based on the extent of the SACK scoreboard. Its goal is to trigger fast retransmit as soon as the receiver's reassembly queue is larger than the dupack threshold, as indicated by the difference between the forward most SACK block edge and SND.UNA. This algorithm quickly and reliably triggers fast retransmit in the presence of burst losses -- often on the first SACK following such a loss. Such a threshold based algorithm also triggers fast retransmit immediately in the presence of any reordering with extent greater than the dupack threshold.

FACK threshold based recovery works by introducing a new TCP state variable at the sender called SND.FACK. SND.FACK reflects the forward-most data held by the receiver and is updated when a SACK block is received acknowledging data with a higher sequence number than the current value of SND.FACK. SND.FACK reflects the highest sequence number known to have been received plus one. Note that in non-recovery states, SND.FACK is the same as SND.UNA. The following snippet is the pseudocode for FACK threshold based recovery.

```
If (SND.FACK - SND.UNA) > dupack threshold:
  -> Invoke Fast Retransmit and Fast Recovery.
```

3. Detecting recovered losses

If the only loss was the last segment, there is the risk that the loss probe itself might repair the loss, effectively masking it from congestion control. To avoid interfering with mandatory congestion control [RFC5681] it is imperative that TLP include a mechanism to detect when the probe might have masked a loss and to properly reduce the congestion window (cwnd). An algorithm to examine subsequent ACKs to determine whether the original segment was lost is described here.

Since it is observed that a significant fraction of the hosts that support SACK do not support duplicate selective acknowledgments (D-SACKs) [RFC2883] the TLP algorithm for detecting such lost segments relies only on basic RFC 2018 SACK [RFC2018].

3.1. TLP Loss Detection: The Basic Idea

Consider a TLP retransmission "episode" where a sender retransmits N consecutive TLP packets, all for the same tail packet in a flight. Let us say that an episode ends when the sender receives an ACK above the SND.NXT at the time of the episode. We want to make sure that before the episode ends the sender receives N "TLP dupacks", indicating that all N TLP probe segments were unnecessary, so there was no loss/hole that needed plugging. If the sender gets less than N "TLP dupacks" before the end of the episode, then probably the first TLP packet to arrive at the receiver plugged a hole, and only the remaining TLP packets that arrived at the receiver generated dupacks.

Note that delayed ACKs complicate the picture, since a delayed ACK will imply that the sender receives one fewer ACK than would normally be expected. To mitigate this complication, before sending a TLP loss probe retransmission, the sender should attempt to wait long enough that the receiver has sent any delayed ACKs that it is withholding. The sender algorithm, described in section 2.1 features such a delay.

If there is ACK loss or a delayed ACK, then this algorithm is conservative, because the sender will reduce cwnd when in fact there was no packet loss. In practice this is acceptable, and potentially even desirable: if there is reverse path congestion then reducing cwnd is prudent.

3.2. TLP Loss Detection: Algorithm Details

(1) State

TLPRtxOut: the number of unacknowledged TLP retransmissions in current TLP episode. The connection maintains this integer counter that tracks the number of TLP retransmissions in the current episode for which we have not yet received a "TLP dupack". The sender initializes the TLPRtxOut field to 0.

TLPHighRxt: the value of SND.NXT at the time of TLP retransmission. The TLP sender uses TLPHighRxt to record SND.NXT at the time it starts doing TLP transmissions during a given TLP episode.

(2) Initialization

When a connection enters the ESTABLISHED state, or suffers a retransmission timeout, or enters fast recovery, it executes the following:

```
TLPRtxOut = 0;
TLPHighRxt = 0;
```

(3) Upon sending a TLP retransmission:

```
if (TLPRtxOut == 0)
      TLPHighRxt = SND.NXT;
TLPRtxOut++;
```

(4) Upon receiving an ACK:

(a) Tracking ACKs

We define a "TLP dupack" as a dupack that has all the regular properties of a dupack that can trigger fast retransmit, plus the ACK acknowledges TLPHighRxt, and the ACK carries no new SACK information (as noted earlier, TLP requires that the receiver supports SACK). This is the kind of ACK we expect to see for a TLP transmission if there were no losses. More precisely, the TLP sender considers a TLP probe segment as acknowledged if all of the following conditions are met:

(a) TLPRtxOut > 0
(b) SEG.ACK == TLPHighRxt
(c) the segment contains no SACK blocks for sequence ranges above TLPHighRxt
(d) the ACK does not advance SND.UNA
(e) the segment contains no data
(f) the segment is not a window update

If all of those conditions are met, then the sender executes the following:

```
TLPRtxOut--;
```

(b) Marking the end of a TLP episode and detecting losses

If an incoming ACK is after TLPHighRxt, then the sender deems the TLP episode over. At that time, the TLP sender executes the following:

```
isLoss = (TLPRtxOut > 0) &&
         (segment does not carry a DSACK for TLP retransmission);
TLPRtxOut = 0
if (isLoss)
 EnterRecovery();
```

In other words, if the sender detects an ACK for data beyond the TLP loss probe retransmission then (in the absence of reordering on the return path of ACKs) it should have received any ACKs that will indicate whether the original or any loss probe retransmissions were lost. An exception is the case when the segment carries a Duplicate SACK (DSACK) for the TLP retransmission. If the TLPRtxOut count is still non-zero and thus indicates that some TLP probe segments remain unacknowledged, then the sender should presume that at least one segment was lost, so it should enter fast recovery using the proportional rate reduction algorithm [IMC11PRR].

(5) Senders must only send a TLP loss probe retransmission if all the conditions from section 2.1 are met and the following condition also holds:

```
(TLPRtxOut == 0) || (SND.NXT == TLPHighRxt)
```

This ensures that there is at most one sequence range with outstanding TLP retransmissions. The sender maintains this invariant so that there is at most one TLP retransmission "episode" happening at a time, so that the sender can use the algorithm described above in this section to determine when the episode is over, and thus when it can infer whether any data segments were lost.

Note that this condition only limits the number of outstanding TLP loss probes that are retransmissions. There may be an arbitrary number of outstanding unacknowledged TLP loss probes that consist of new, previously-unsent data, since the standard retransmission timeout and fast recovery algorithms are sufficient to detect losses of such probe segments.

4. Discussion

In this section we discuss two properties related to TLP.

4.1. Unifying loss recoveries

The existing loss recovery algorithms in TCP have a discontinuity: A single segment loss in the middle of a packet train can be recovered via fast recovery while a loss at the end of the train causes an RTO. Example: consider a train of segments 1-10, loss of segment five can be recovered quickly through fast recovery, while loss of segment ten can only be recovered through a timeout. In practice, the difference between losses that trigger RTO versus those invoking fast recovery has more to do with the position of the losses as opposed to the intensity or magnitude of congestion at the link.

TLP unifies the loss recovery mechanisms regardless of the position of a loss, so now with TLP a segment loss in the middle of a train as well as at the tail end can now trigger the same fast recovery mechanisms.

4.2. Recovery of any N-degree tail loss

The TLP algorithm, when combined with a variant of the early retransmit mechanism described below, is capable of recovering any tail loss for any sized flow using fast recovery.

We propose the following enhancement to the early retransmit algorithm described in [RFC5827]: in addition to allowing an early retransmit in the scenarios described in [RFC5827], we propose to allow a delayed early retransmit [IMC11PRR] in the case where there are three outstanding segments that have not been cumulatively acknowledged and one segment that has been fully SACKed.

Consider the following scenario, which illustrates an example of how this enhancement allows quick loss recovery in a new scenario:

```
(1) scoreboard reads: A _ _ _
(2) TLP retransmission probe of the last (fourth) segment
(3) the arrival of a SACK for the last segment changes
    scoreboard to: A _ _ S
(4) early retransmit and fast recovery of the second and
    third segments
```

With this enhancement to the early retransmit mechanism, then for any degree of N-segment tail loss we get a quick recovery mechanism instead of an RTO.

Consider the following taxonomy of tail loss scenarios, and the ultimate outcome in each case:

	number of losses	scoreboard after TLP retrans ACKed	mechanism	final outcome
(1)	AAAL	AAAA	TLP loss detection	all repaired
(2)	AALL	AALS	early retransmit	all repaired
(3)	ALLL	ALLS	early retransmit	all repaired
(4)	LLLL	LLLS	FACK fast recovery	all repaired
(5)	>=5 L	..LS	FACK fast recovery	all repaired

key:
A = ACKed segment
L = lost segment
S = SACKed segment

Let us consider each tail loss scenario in more detail:

(1) With one segment lost, the TLP loss probe itself will repair the loss. In this case, the sender's TLP loss detection algorithm will notice that a segment was lost and repaired, and reduce its congestion window in response to the loss.

(2) With two segments lost, the TLP loss probe itself is not enough to repair the loss. However, when the SACK for the loss probe arrives at the sender, then the early retransmit mechanism described in [RFC5827] will note that with two segments outstanding and the second one SACKed, the sender should retransmit the first segment. This retransmit will repair the single remaining lost segment.

(3) With three segments lost, the TLP loss probe itself is not enough to repair the loss. However, when the SACK for the loss probe arrives at the sender, then the enhanced early retransmit mechanism described in this section will note that with three segments outstanding and the third one SACKed, the sender should retransmit the first segment and enter fast recovery. The early retransmit and fast recovery phase will, together, repair the the remaining two lost segments.

(4) With four segments lost, the TLP loss probe itself is not enough to repair the loss. However, when the SACK for the loss probe arrives at the sender, then the FACK fast retransmit mechanism [FACK] will note that with four segments outstanding and the fourth one SACKed, the sender should retransmit the first segment and enter fast recovery. The fast retransmit and fast recovery phase will, together, repair the the remaining two lost segments.

(5) With five or more segments lost, events precede much as in case (4). The TLP loss probe itself is not enough to repair the loss.

However, when the SACK for the loss probe arrives at the sender, then the FACK fast retransmit mechanism [FACK] will note that with five or more segments outstanding and the segment highest in sequence space SACKed, the sender should retransmit the first segment and enter fast recovery. The fast retransmit and fast recovery phase will, together, repair the remaining lost segments.

In summary, the TLP mechanism, in conjunction with the proposed enhancement to the early retransmit mechanism, is able to recover from a tail loss of any number of segments without resort to a costly RTO.

5. Experiments with TLP

In this section we describe experiments and measurements with TLP performed on Google Web servers using Linux 2.6. The experiments were performed over several weeks and measurements were taken across a wide range of Google applications. The main goal of the experiments is to instrument and measure TLP's performance relative to the baseline. The experiment and baseline were using the same kernels with an on/off switch to enable TLP.

Our experiments include both the basic TLP algorithm of Section 2 and its loss detection component in Section 3. All other algorithms such as early retransmit and FACK threshold based recovery are present in the both the experiment and baseline. There are three primary metrics we are interested in: impact on TCP latency (average and tail or 99th percentile latency), retransmission statistics, and the overhead of probe segments relative to the total number of transmitted segments. TCP latency is the time elapsed between the server transmitting the first byte of the response to it receiving an ACK for the last byte.

The table below shows the percentiles and average latency improvement of key Web applications, including even those responses without losses, measured over a period of one week. The key takeaway is: the average response time improved up to 7% and the 99th percentile improved by 10%. Nearly all of the improvement for TLP is in the tail latency (post-90th percentile). The varied improvements across services are due to different response-size distributions and traffic patterns. For example, TLP helps the most for Images, as these are served by multiple concurrently active TCP connections which increase the chances of tail segment losses.

Application	Average	99%
Google Web Search	-3%	-5%
Google Maps	-5%	-10%
Google Images	-7%	-10%

TLP also improved performance in mobile networks -- by 7.2% for Web search and Instant and 7.6% for Images transferred over Verizon network. To see why and where the latency improvements are coming from, we measured the retransmission statistics. We broke down the retransmission stats based on nature of retransmission -- timeout retransmission or fast recovery. TLP reduced the number of timeouts by 15% compared to the baseline, i.e. (timeouts_tlp - timeouts_baseline) / timeouts_baseline = 15%. Instead, these losses were either recovered via fast recovery or by the loss probe retransmission itself. The largest reduction in timeouts is when the sender is in the Open state in which it receives only insequence ACKs and no duplicate ACKs, likely because of tail losses. Correspondingly, the retransmissions occurring in the slow start phase after RTO reduced by 46% relative to baseline. Note that it is not always possible for TLP to convert 100% of the timeouts into fast recovery episodes because a probe itself may be lost. Also notable in our experiments is a significant decrease in the number of spurious timeouts -- the experiment had 61% fewer congestion window undo events. The Linux TCP sender uses either DSACK or timestamps to determine if retransmissions are spurious and employs techniques for undoing congestion window reductions. We also note that the total number of retransmissions decreased 7% with TLP because of the decrease in spurious retransmissions, and because the TLP probe itself plugs a hole.

We also quantified the overhead of probe packets. The probes accounted for 0.48% of all outgoing segments, i.e. (number of probe segments / number of outgoing segments)*100 = 0.48%. This is a reasonable overhead when contrasted with the overall retransmission rate of 3.2%. 10% of the probes sent are new segments and the rest are retransmissions, which is unsurprising given that short Web responses often don’t have new data to send. We also found that in about 33% of the cases, the probes themselves plugged the only hole at receiver and the loss detection algorithm reduced the congestion window. 37% of the probes were not necessary and resulted in a duplicate acknowledgment.

Besides the macro level latency and retransmission statistics, we report some measurements from TCP’s internal state variables at the point when a probe segment is transmitted. The following distribution shows the FlightSize and congestion window values when a PTO is scheduled. We note that cwnd is not the limiting factor and that nearly all of the probe segments are sent within the congestion window.

percentile	10%	25%	50%	75%	90%	99%
FlightSize	1	1	2	3	10	20
cwnd	5	10	10	10	17	44

We have also experimented with a few variations of TLP: multiple probe segments versus single probe for the same tail loss episode, and several values for WCDelAckT. Our experiments show that sending just one probe suffices to get most (~90%) of latency benefits. The experiment results reported in this section and our current implementation limits number of probes to one, although the draft itself allows up to two consecutive probes. We chose the worst case delayed ack timer to be 200ms. When FlightSize equals 1 it is important to account for the delayed ACK timer in the PTO value, in order to bring down the number of unnecessary probe segments. With delays of 0ms and 50ms, the probe overhead jumped from 0.48% to 3.1% and 2.2% respectively. We have also experimented with transmitting 1-byte probe retransmissions as opposed to an entire MSS retransmission probe. While this scheme has the advantage of not requiring the loss detection algorithm outlined in Section 3, it turned out to be problematic to implement correctly in certain TCP stacks. Additionally, retransmitting 1-byte probe costs one more RTT to recover single packet tail losses, which is detrimental for short transfer latency.

6. Related work

TCP's long and conservative RTO recovery has long been identified as the major performance bottleneck for latency-demanding applications. A well-studied example is online gaming that requires reliability and low latency but small bandwidth. [GRIWODZ06] shows that repeated long RTO is the dominating performance bottleneck for game responsiveness. The authors in [PETLUND08] propose to use linear RTO to improve the performance, which has been incorporated in the Linux kernel as a non-default socket option for such thin streams. [MONDAL08] further argues exponential RTO backoff should be removed because it is not necessary for the stability of Internet. In contrast, TLP does not change the RTO timer calculation or the exponential back off. TLP's approach is to keep the behavior after RTO conservative for stability but allows a few timely probes before concluding the network is badly congested and cwnd should fall to 1.

As noted earlier in the Introduction the F-RTO [RFC5682] algorithm reduces the number of spurious timeout retransmissions and the Early Retransmit [RFC5827] mechanism reduces timeouts when a connection has received a certain number of duplicate ACKs. Both are complementary to TLP and can work alongside. Rescue retransmission introduced in [RFC6675] deals with loss events such as AL*SL* (using the same notation as section 4). TLP covers wider range of events such as AL*. We experimented with rescue retransmission on Google Web servers, but did not observe much performance improvement. When the last segment is lost, it is more likely that a number of contiguous segments preceding the segment are also lost, i.e. AL* is common. Timeouts that occur in the fast recovery are rare.

[HURTIG13] proposes to offset the elapsed time of the pending packet when re-arming the RTO timer. It is possible to apply the same idea for the TLP timer as well. We have not yet tested such a change to TLP.

Tail Loss Probe is one of several algorithms designed to maximize the robustness of TCPs self clock in the presence of losses. It follows the same principles as Proportional Rate Reduction [IMC11PRR] and TCP Laminar [Laminar].

On a final note we note that Tail loss probe does not eliminate 100% of all RTOs. RTOs still remain the dominant mode of loss recovery for short transfers. More work in future should be done along the following lines: transmitting multiple loss probes prior to finally resorting to RTOs, maintaining ACK clocking for short transfers in the absence of new data by clocking out old data in response to incoming ACKs, taking cues from applications to indicate end of transactions and use it for smarter tail loss recovery.

7. Security Considerations

The security considerations outlined in [RFC5681] apply to this document. At this time we did not find any additional security problems with Tail loss probe.

8. IANA Considerations

This document makes no request of IANA.

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

9. References

[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.

[RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, June 2011.

[RFC5827] Allman, M., Ayesta, U., Wang, L., Blanton, J., and P. Hurtig, "Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)", RFC 5827, April 2010.

[RFC5682] Sarolahti, P., Kojo, M., Yamamoto, K., and M. Hata, "Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP", RFC 5682, September 2009.

[IMC11PRR] Mathis, M., Dukkipati, N., Cheng, Y., and M. Ghobadi, "Proportional Rate Reduction for TCP", Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference , 2011.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, March 1997.

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

[FACK] Mathis, M. and M. Jamshid, "Forward acknowledgement: refining TCP congestion control", ACM SIGCOMM Computer Communication Review, Volume 26, Issue 4, Oct. 1996. , 1996.

[RFC2883] Floyd, S., Mahdavi, J., Mathis, M., and M. Podolsky, "An Extension to the Selective Acknowledgement (SACK) Option for TCP", RFC 2883, July 2000.

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

[GRIWODZ06] Griwodz, C. and P. Halvorsen, "The fun of using TCP for an MMORPG", NOSSDAV , 2006.

[PETLUND08]
Petlund, A., Evensen, K., Griwodz, C., and P. Halvorsen, "TCP enhancements for interactive thin-stream applications", NOSSDAV , 2008.

[MONDAL08]
Mondal, A. and A. Kuzmanovic, "Removing Exponential Backoff from TCP", ACM SIGCOMM Computer Communication Review , 2008.

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

[HURTIG13]
Hurtig, P., Brunstrom, A., Petlund, A., and M. Welzl, "TCP and SCTP RTO Restart", draft-ietf-tcpm-rtorestart-00 (work in progress), February 2013.

Authors' Addresses

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

Email: nanditad@google.com

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

Email: ncardwell@google.com

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

Email: ycheng@google.com

Matt Mathis
Google, Inc
1600 Amphitheater Parkway
Mountain View, California 93117
USA

Email: mattmathis@google.com