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 3
  1.1. Terminology 5
2. Loss probe algorithm 5
  2.1. Pseudocode 6
  2.2. FACK threshold based recovery 8
3. Detecting recovered losses 9
  3.1. TLP Loss Detection: The Basic Idea 9
  3.2. TLP Loss Detection: Algorithm Details 9
4. Discussion 11
  4.1. Unifying loss recoveries 12
  4.2. Recovery of any N-degree tail loss 12
5. Experiments with TLP 14
6. Related work 16
7. Security Considerations 17
8. IANA Considerations 17
9. References 18
Authors' Addresses 19

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

TCP Maintenance and Minor Extensions
(tcpm)
Internet-Draft
Updates: 793 (if approved)
Intended status: Standards Track
Expires: August 20, 2013

F. Gont
UTN-FRH / SI6 Networks
D. Borman
Quantum Corporation
February 16, 2013

On the Validation of TCP Sequence Numbers
draft-gont-tcpm-tcp-seq-validation-00.txt

Abstract

When TCP receives packets that lie outside of the receive window, the corresponding packets are dropped and either an ACK, RST or no response is generated due to the out-of-window packet, with no further processing of the packet. Most of the time, this works just fine and TCP remains stable, especially when a TCP connection has unidirectional data flow. However, there are three scenarios in which packets that are outside of the receive window should still have their ACK field processed, or else a packet war will take place. The aforementioned issues have affected a number of popular TCP implementations, typically leading to connection failures, system crashes, or other undesirable behaviors. This document describes the three scenarios in which the aforementioned issues might arise, and formally updates RFC 793 such that these potential problems are mitigated.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction 3
2. TCP Sequence Number Validation 3
3. Scenarios in which Undesirable Behaviors Might Arise 4
3.1. TCP simultaneous open 4
3.2. TCP self connects 5
3.3. TCP simultaneous close 6
3.4. Simultaneous Window Probes 8
4. Updating RFC 793 9
4.1. TCP sequence number validation 9
4.2. TCP self connects 14
5. IANA Considerations 14
6. Security Considerations 14
7. Acknowledgements 14
8. References 14
8.1. Normative References 14
8.2. Informative References 15
Authors' Addresses 15

1. Introduction

TCP processes incoming packets in in-sequence order. Packets that are not in-sequence but have data that lies in the receive window are queued for later processing. Packets that lie outside of the receive window are dropped and either an ACK, RST or no response is generated due to the out-of-window packet, with no further processing of the packet. Most of the time, this works just fine and TCP remains stable, especially when a TCP connection has unidirectional data flow.

However, there are three situations in which packets that are outside of the receive window should still have their ACK field processed. These situations arise during a simultaneous open, simultaneous window probes and a simultaneous close. In all three of these cases, a packet will arrive with a sequence number that is one to the left of the window, but the acknowledgement field has updated information that needs to be processed to avoid entering a packet war, in which both sides of the connection generate a response to the received packet, which just causes the other side to do the same thing. This issue has affected a number of popular TCP implementations, typically leading to connection failures, system crashes, or other undesirable behaviors.

Section 2 provides an overview of the TCP sequence number validation checks specified in RFC 793. Section 3 describes the three scenarios in which the current TCP sequence number validation checks can lead to undesirable behaviors. Section 4 formally updates RFC 793 such that these issues are mitigated.

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 RFC 2119 [RFC2119].

2. TCP Sequence Number Validation

Section 3.3 of RFC 793 [RFC0793] specifies (in pp. 25-26) how the TCP sequence number of incoming segments is to be validated. It summarizes the validation of the TCP sequence number with the following table:

```
Segment Receive  Test
Length  Window
------- -------  -------------------------------------------

   0       0     SEG.SEQ = RCV.NXT

   0      >0     RCV.NXT =< SEG.SEQ < RCV.NXT+RCV.WND

  >0       0     not acceptable

  >0      >0     RCV.NXT =< SEG.SEQ < RCV.NXT+RCV.WND
              or RCV.NXT =< SEG.SEQ+SEG.LEN-1 < RCV.NXT+RCV.WND
```

RFC 793 states that if an incoming segment is not acceptable, an acknowledgment should be sent in reply (unless the RST bit is set), and that after sending the acknowledgment, the unacceptable segment should be dropped.

Section 3.9 of RFC 793 repeats (in pp. 69-76) the same validation checks when describing the processing of incoming TCP segments meant for connections that are in the SYN-RECEIVED, ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, or TIME-WAIT states (i.e., any state other than CLOSED, LISTEN, or SYN-SENT).

A key problem with the aforementioned checks is that it assumes that a segment must be processed only if a portion of it overlaps with the receive window. However, there are some cases in which the Acknowledgement information in an incoming segment needs to be processed by TCP even if the contents of the segment does not overlap with the receive window. Otherwise, the TCP state machine may become dead-locked, and this situation may result in undesirable behaviors such as system crashes.

3. Scenarios in which Undesirable Behaviors Might Arise

The following subsections describe the three scenarios in which the TCP Sequence Number validation specified n RFC 793 (and described in Section 2 of this document) could result in undesirable behaviors.

3.1. TCP simultaneous open

The following figure illustrates a typical "simultaneous open" attempt.

```
      TCP A                                                  TCP B

   1. CLOSED                                                 CLOSED

   2. SYN-SENT     --> <SEQ=100><CTL=SYN>              ...

   3. SYN-RECEIVED <-- <SEQ=300><CTL=SYN>              <-- SYN-SENT

   4.              ... <SEQ=100><CTL=SYN>              --> SYN-RECEIVED

   5.              --> <SEQ=100><ACK=301><CTL=SYN,ACK> ...

   6.              <-- <SEQ=300><ACK=101><CTL=SYN,ACK> <--

   7.              ... <SEQ=100><ACK=301><CTL=SYN,ACK>  -->

   8.              --> <SEQ=100><ACK=301><CTL=SYN,ACK> ...

   9.              <-- <SEQ=300><ACK=101><CTL=SYN,ACK> <--

   10.             ... <SEQ=100><ACK=301><CTL=ACK>     -->
```

(Failed) Simultaneous Connection Synchronization

In line 2, TCP A performs an "active open" by sending a SYN segment to TCP B, and enters the SYN-SENT state. In line 3, TCP B performs an "active open" by sending a SYN segment to TCP A, and enters the "SYN-SENT" state; when TCP A receives this SYN segment sent by TCP B, it enters the SYN-RECEIVED state, and its RCV.NXT becomes 301. In line 4, similarly, when TCP B receives the SYN segment sent by TCP A, it enters the SYN-RECEIVED STATE and its RCV.NXT becomes 101. In line 5, TCP A sends a SYN/ACK in response to the received SYN segment from line 3. In line 6, TCP B sends a SYN/ACK in response to the received SYN segment from line 4. In line 7, TCP B receives the SYN/ACK from line 5. In line 8, TCP A receives the SYN/ACK from line 6, which fails the TCP Sequence Number validation check. As a result, the received packet is dropped, and a SYN/ACK is sent in response. In line 9, TCP B processes the SYN/ACK from line 7, which fails the TCP Sequence Number validation check. As a result, the received packet is dropped, and a SYN/ACK is sent in response. In line 10, the SYN/ACK from line 9 arrives at TCP B. The segment exchange from lines 8-10 will continue forever (with both TCP end-points will be stuck in the SYN-RECEIVED state), thus leading to a SYN/ACK war.

3.2. TCP self connects

Some systems have been found to be unable to process TCP connection requests in which the source endpoint {Source Address, Source Port}

is the same as the destination end-point {Destination Address, Destination Port}. Such a scenario might arise e.g. if a process creates a socket, bind()s a local end-point (IP address and TCP port), and then issues a connect() to the same end-point as that specified to bind().

While not widely employed in existing applications, such a socket could be employed as a "full-duplex pipe" for Inter-Process Communication (IPC).

This scenario is described in detail in pp. 960-962 of [Wright1994].

The aforementioned scenario has been reported to cause malfunction of a number of implementations [CERT1996], and has been exploited in the past to perform Denial of Service (DoS) attacks [Meltman1997] [CPNI-TCP].

While this scenario is not common in the real world, TCP should nevertheless be able to process them without the need of any "extra" code: a SYN segment in which the source end-point {Source Address, Source Port} is the same as the destination end-point {Destination Address, Destination Port} should result in a "simultaneous open" scenario, such as the one described in page 32 of RFC 793 [RFC0793]. Therefore, those TCP implementations that correctly handle simultaneous opens should already be prepared to handle these unusual TCP segments.

3.3. TCP simultaneous close

The following figure illustrates a typical "simultaneous close" attempt, in which the FIN segments sent by each TCP end-point cross each other in the network.

```
      TCP A                                                  TCP B

   1. ESTABLISHED                                            ESTABLISHED

   2. FIN-WAIT-1   --> <SEQ=100><ACK=300><CTL=FIN,ACK> ...

   3. CLOSING      <-- <SEQ=300><ACK=100><CTL=FIN,ACK> <-- FIN-WAIT-1

   4.              ... <SEQ=100><ACK=300><CTL=FIN,ACK> --> CLOSING

   5.              --> <SEQ=100><ACK=301><CTL=FIN,ACK> ...

   6.              <-- <SEQ=300><ACK=101><CTL=FIN,ACK> <--

   7.              ... <SEQ=100><ACK=301><CTL=FIN,ACK> -->

   8.              --> <SEQ=100><ACK=301><CTL=FIN,ACK> ...

   9.              <-- <SEQ=300><ACK=101><CTL=FIN,ACK> <--

   10.             ... <SEQ=100><ACK=301><CTL=FIN,ACK> -->
```

(Failed) Simultaneous Connection Termination

In line 1, we assume that both end-points of the connection are in the ESTABLISHED state. In line 2, TCP A performs an "active close" by sending a FIN segment to TCP B, thus entering the FIN-WAIT-1 state. In line 3, TCP B performs an active close sending a FIN segment to TCP A, thus entering the FIN-WAIT-1 state; when this segment is processed by TCP A, it enters the CLOSING state (and its RCV.NXT becomes 301).

Both FIN segments cross each other on the network, thus resulting in a "simultaneous connection termination" (or "simultaneous close") scenario.

In line 4, the FIN segment sent by TCP A arrives to TCP B, causing it to transition to the CLOSING state (at this point, TCP B's RCV.NXT becomes 101). In line 5, TCP A acknowledges the receipt of the TCP B's FIN segment, and also sets the FIN bit in the outgoing segment (since it has not yet been acknowledged). In line 6, TCP B acknowledges the receipt of TCP A's FIN segment, and also sets the FIN bit in the outgoing segment (since it has not yet been acknowledged). In line 7, the FIN/ACK from line 5 arrives at TCP B. In line 8, the FIN/ACK from line 6 fails the TCP sequence number validation check, and thus elicits a ACK segment (the segment also contains the FIN bit set, since it had not yet been acknowledged). In line 9, the FIN/ACK from line 7 fails the TCP sequence number

validation check, and hence elicits an ACK segment (the segment also contains the FIN bit set, since it had not yet been acknowledged). In line 10, the FIN/ACK from line 8 finally arrives at TCP B.

The packet exchange from lines 8-10 will repeat indefinitely, with both TCP end-points stuck in the CLOSING state, thus leading to a "FIN war": each FIN/ACK segment sent by a TCP will elicit a FIN/ACK from the other TCP, and each of these FIN/ACKs will in turn elicit more FIN/ACKs.

3.4. Simultaneous Window Probes

The following figure illustrates a scenario in which the "persist timer" at both TCP end-points expires, and both TCP end-points send a "window probes" that cross each other in the network.

```
   TCP A                                                TCP B

1. ESTABLISHED                                          ESTABLISHED

2.                      (both TCP windows open)

3.            --> <SEQ=100><DATA=1><ACK=300><CTL=ACK> ...

4.            <-- <SEQ=300><DATA=1><ACK=100><CTL=ACK> <--

5.            ... <SEQ=100><DATA=1><ACK=300><CTL=ACK> -->

6.            --> <SEQ=100><ACK=301><CTL=ACK>         ...

7.            <-- <SEQ=300><ACK=101><CTL=ACK>         <--

8.            ... <SEQ=100><ACK=301><CTL=ACK>         -->

9.            --> <SEQ=100><ACK=301><CTL=ACK>         ...

10.           <-- <SEQ=300><ACK=101><CTL=ACK>         <--

11.           ... <SEQ=100><ACK=301><CTL=ACK>         -->

              (Failed) Simultaneous Connection Termination
```

In line 1, we assume that both end-points of the connection are in the ESTABLISHED state; additionally, TCP A’s RCV.NXT is 300, while TCP B’s RCV.NXT is 100, and the receive window (RCV.WND) at both TCP end-points is 0. In line 2, both TCP windows open. In line 3, the "persist timer" at TCP A expires, and hence TCP A sends a "Window Probe". In line 4, the "persist timer" at TCP B expires, and hence

TCP B sends a "Window Probe".

Both Window Probes cross each other in the network.

When this probe arrives at TCP A, TCP a's RCV.NXT becomes 301, and an ACK segment is sent to advertise the new window (this ACK is shown in line 6). In line 5, TCP A's Window Probe from line 3 arrives at TCP B. TCP B's RCV-WND becomes 101. In line 6, TCP A sends the ACK to advertise the new window. In line 7, TCP B sends an ACK to advertise the new Window. When this ACK arrives at TCP A, the TCP Sequence Number validation fails, since SEG.SEQ=300 and RCV.NXT=301. Therefore, this segment elicits a new ACK (meant to re-synchronize the sequence numbers). In line 8, the ACK from line 6 arrives at TCP B. The TCP sequence number validation for this segment fails, since SEG.SEQ=100 AND RCV.NXT=101. Therefore, this segment elicits a new ACK (meant to re-synchronize the sequence numbers).

Line 9 and line 11 shows the ACK elicited by the segment from line 7, while line 10 shows the ACK elicited by the segment from line 8. The sequence numbers of these ACK segments will be considered invalid, and hence will elicit further ACKs. Therefore, the segment exchange from lines 9-11 will repeat indefinitely, thus leading to an "ACK war".

4. Updating RFC 793

4.1. TCP sequence number validation

The following text from Section 3.3 (pp. 25-26) of [RFC0793]:

A segment is judged to occupy a portion of valid receive sequence space if

```
RCV.NXT =< SEG.SEQ < RCV.NXT+RCV.WND
```

or

```
RCV.NXT =< SEG.SEQ+SEG.LEN-1 < RCV.NXT+RCV.WND
```

The first part of this test checks to see if the beginning of the segment falls in the window, the second part of the test checks to see if the end of the segment falls in the window; if the segment passes either part of the test it contains data in the window.

Actually, it is a little more complicated than this. Due to zero windows and zero length segments, we have four cases for the acceptability of an incoming segment:

```
Segment Receive  Test
Length  Window
------- -------  -------------------------------------------

   0       0     SEG.SEQ = RCV.NXT

   0      >0     RCV.NXT =< SEG.SEQ < RCV.NXT+RCV.WND

  >0       0     not acceptable

  >0      >0     RCV.NXT =< SEG.SEQ < RCV.NXT+RCV.WND
                 or RCV.NXT =< SEG.SEQ+SEG.LEN-1 < RCV.NXT+RCV.WND
```

is replaced with:

A segment is judged to occupy a portion of valid receive sequence space if

```
RCV.NXT-1 =< SEG.SEQ < RCV.NXT+RCV.WND
```

or

```
RCV.NXT-1 =< SEG.SEQ+SEG.LEN-1 < RCV.NXT+RCV.WND
```

The first part of this test checks to see if the beginning of the segment falls in the window (or one byte to the left to the window), the second part of the test checks to see if the end of the segment falls in the window (or one byte to the left of the window); if the segment passes either part of the test it contains data in the window or control information that needs to be processed by TCP.

Actually, it is a little more complicated than this. Due to zero windows and zero length segments, we have four cases for the acceptability of an incoming segment:

```
Segment Receive  Test
Length  Window
------- -------  -------------------------------------------

   0       0     RCV.NXT-1 =< SEG.SEQ <= RCV.NXT

   0      >0     RCV.NXT-1 =< SEG.SEQ < RCV.NXT+RCV.WND

  >0       0     not acceptable

  >0      >0     RCV.NXT-1 =< SEG.SEQ < RCV.NXT+RCV.WND
                 or RCV.NXT-1 =< SEG.SEQ+SEG.LEN-1 < RCV.NXT+RCV.WND
```

Additionally, the following text from Section 3.9 (pp.69-70) of [RFC0793]:

Segments are processed in sequence. Initial tests on arrival are used to discard old duplicates, but further processing is done in SEG.SEQ order. If a segment's contents straddle the boundary between old and new, only the new parts should be processed.

There are four cases for the acceptability test for an incoming segment:

```
Segment Receive  Test
Length  Window
------- -------  -------------------------------------------

   0       0     SEG.SEQ = RCV.NXT

   0      >0     RCV.NXT =< SEG.SEQ < RCV.NXT+RCV.WND

  >0       0     not acceptable

  >0      >0     RCV.NXT =< SEG.SEQ < RCV.NXT+RCV.WND
              or RCV.NXT =< SEG.SEQ+SEG.LEN-1 < RCV.NXT+RCV.WND
```

If the RCV.WND is zero, no segments will be acceptable, but special allowance should be made to accept valid ACKs, URGs and RSTs.

If an incoming segment is not acceptable, an acknowledgment should be sent in reply (unless the RST bit is set, if so drop the segment and return):

```
<SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>
```

After sending the acknowledgment, drop the unacceptable segment and return.

In the following it is assumed that the segment is the idealized segment that begins at RCV.NXT and does not exceed the window. One could tailor actual segments to fit this assumption by trimming off any portions that lie outside the window (including SYN and FIN), and only processing further if the segment then begins at RCV.NXT. Segments with higher beginning sequence numbers may be held for later processing.

is replaced with:

Segments are processed in sequence. Initial tests on arrival are used to discard old duplicates, but further processing is done in SEG.SEQ order. If a segment's contents straddle the boundary between old and new, only the new parts should be processed. Acknowledgement information must still be processed when the contents of the incoming segment are one byte to the left of the receive window.

This is to handle simultaneous opens, simultaneous closes, and simultaneous window probes.

There are four cases for the acceptability test for an incoming segment:

```
Segment Receive  Test
Length  Window
------- -------  -------------------------------------------

   0       0     RCV.NXT-1 =< SEG.SEQ <= RCV.NXT

   0      >0     RCV.NXT-1 =< SEG.SEQ < RCV.NXT+RCV.WND

  >0       0     not acceptable

  >0      >0     RCV.NXT-1 =< SEG.SEQ < RCV.NXT+RCV.WND
                 or RCV.NXT-1 =< SEG.SEQ+SEG.LEN-1 < RCV.NXT+RCV.WND
```

If the RCV.WND is zero, no segments will be acceptable, but special allowance should be made to accept valid ACKs, URGs and RSTs.

If an incoming segment is not acceptable, an acknowledgment should be sent in reply (unless the RST bit is set, if so drop the segment and return):

```
<SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>
```

After sending the acknowledgment, drop the unacceptable segment and return.

In the following it is assumed that the segment is the idealized segment that begins at RCV.NXT and does not exceed the window. One could tailor actual segments to fit this assumption by trimming off any portions that lie outside the window (including SYN and FIN). Segments with higher beginning sequence numbers may be held for later processing. Acknowledgement information must still be processed when the contents of the incoming segment are one byte to the left of the receive window.

4.2. TCP self connects

TCP MUST be able to gracefully handle connection requests (i.e., SYN segments) in which the source end-point (IP Source Address, TCP Source Port) is the same as the destination end-point (IP Destination Address, TCP Destination Port). Such segments MUST result in a TCP "simultaneous open", such as the one described in page 32 of RFC 793 [RFC0793].

> Those TCP implementations that correctly handle simultaneous opens are expected to gracefully handle this scenario.

5. IANA Considerations

This document has no IANA actions. The RFC Editor is requested to remove this section before publishing this document as an RFC.

6. Security Considerations

This document describes a problem found in the current validation rules for TCP sequence numbers. The aforementioned problem has affected some popular TCP implementations, typically leads to connection failures, system crashes, or other undesirable behaviors. This document formally updates RFC 793, such that the aforementioned issues are eliminated.

7. Acknowledgements

This document originated from a discussion about this topic (at IETF 73, Minneapolis) between both co-authors of this document.

8. References

8.1. Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

8.2. Informative References

[CERT1996]
CERT, "CERT Advisory CA-1996-21: TCP SYN Flooding and IP Spoofing Attacks", 1996, <http://www.cert.org/advisories/CA-1996-21.html>.

[CPNI-TCP]
Gont, F., "CPNI Technical Note 3/2009: Security Assessment of the Transmission Control Protocol (TCP)", 2009, <http://www.gont.com.ar/papers/tn-03-09-security-assessment-TCP.pdf>.

[Meltman1997]
Meltman, "new TCP/IP bug in win95. Post to the bugtraq mailing-list", 1996, <http://insecure.org/sploits/land.ip.DOS.html>.

[Wright1994]
Wright, G. and W. Stevens, "TCP/IP Illustrated, Volume 2: The Implementation", Addison-Wesley, 1994.

Authors' Addresses

Fernando Gont
UTN-FRH / SI6 Networks
Evaristo Carriego 2644
Haedo, Provincia de Buenos Aires 1706
Argentina

Phone: +54 11 4650 8472
Email: fgont@si6networks.com
URI: http://www.si6networks.com

David Borman
Quantum Corporation
1155 Centre Pointe Drive, Suite 1
Mendota Heights, MN 55120
U.S.A.

Phone: 651-688-4394
Email: david.borman@quantum.com

TCP Maintenance (TCPM)
Internet-Draft
Intended status: Standards Track
Expires: August 29, 2013

D. Borman
Quantum Corporation
B. Braden
University of Southern California
V. Jacobson
Packet Design
R. Scheffenegger, Ed.
NetApp, Inc.
February 25, 2013

TCP Extensions for High Performance
draft-ietf-tcpm-1323bis-06

Abstract

This document specifies a set of TCP extensions to improve performance over paths with a large bandwidth*delay product and to provide reliable operation over very high-speed paths. It defines TCP options for scaled windows and timestamps. The timestamps are used for two distinct mechanisms, RTTM (Round Trip Time Measurement) and PAWS (Protection Against Wrapped Sequences).

This document updates and obsoletes RFC 1323.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction 4
1.1. TCP Performance 4
1.2. TCP Reliability 5
1.3. Using TCP options 6
2. Terminology 7
3. TCP Window Scale Option 7
3.1. Introduction 7
3.2. Window Scale Option 7
3.3. Using the Window Scale Option 8
3.4. Addressing Window Retraction 10
4. RTTM -- Round-Trip Time Measurement 11
4.1. Introduction 11
4.2. TCP Timestamps Option 12
4.3. The RTTM Mechanism 13
4.4. Which Timestamp to Echo 14
5. PAWS -- Protection Against Wrapped Sequence Numbers 17
5.1. Introduction 17
5.2. The PAWS Mechanism 17
5.2.1. Basic PAWS Algorithm 18
5.2.2. Timestamp Clock 20
5.2.3. Outdated Timestamps 22
5.2.4. Header Prediction 22
5.2.5. IP Fragmentation 24
5.3. Duplicates from Earlier Incarnations of Connection 24
6. Conclusions and Acknowledgements 24
7. Security Considerations 25
8. IANA Considerations 26
9. References 26
9.1. Normative References 26
9.2. Informative References 26
Appendix A. Implementation Suggestions 28
Appendix B. Duplicates from Earlier Connection Incarnations . . . 29
B.1. System Crash with Loss of State 30
B.2. Closing and Reopening a Connection 30
Appendix C. Summary of Notation 31
Appendix D. Pseudo-code Summary 32
Appendix E. Event Processing Summary 34
Appendix F. Timestamps Edge Cases 40
Appendix G. Changes from RFC 1072, RFC 1185, and RFC 1323 40
Authors' Addresses 43

1. Introduction

The TCP protocol [RFC0793] was designed to operate reliably over almost any transmission medium regardless of transmission rate, delay, corruption, duplication, or reordering of segments. Over the years, advances in networking technology has resulted in ever-higher transmission speeds, and the fastest paths are well beyond the domain for which TCP was originally engineered.

This document defines a set of modest extensions to TCP to extend the domain of its application to match the increasing network capability. It is an update to and obsoletes [RFC1323], which in turn is based upon and obsoletes [RFC1072] and [RFC1185].

For brevity, the full discussions of the merits and history behind the TCP options defined within this document have been omitted. [RFC1323] should be consulted for reference. A modern TCP implementation SHOULD implement and make use of the extensions described in this document.

1.1. TCP Performance

TCP performance problems arise when the bandwidth*delay product is large. A network having such paths is referred to as "long, fat network" (LFN).

There are three fundamental performance problems with the current TCP over LFN paths:

(1) Window Size Limit

The TCP header uses a 16 bit field to report the receive window size to the sender. Therefore, the largest window that can be used is 2^16 = 65K bytes.

To circumvent this problem, Section 2 of this memo defines a new TCP option, "Window Scale", to allow windows larger than 2^16. This option defines an implicit scale factor, which is used to multiply the window size value found in a TCP header to obtain the true window size.

(2) Recovery from Losses

Packet losses in an LFN can have a catastrophic effect on throughput.

To generalize the Fast Retransmit/Fast Recovery mechanism to handle multiple packets dropped per window, selective

acknowledgments are required. Unlike the normal cumulative acknowledgments of TCP, selective acknowledgments give the sender a complete picture of which segments are queued at the receiver and which have not yet arrived.

Selective acknowledgements are specified in a separate document, "A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP" [RFC6675], and not further discussed in this document.

(3) Round-Trip Measurement

TCP implements reliable data delivery by retransmitting segments that are not acknowledged within some retransmission timeout (RTO) interval. Accurate dynamic determination of an appropriate RTO is essential to TCP performance. RTO is determined by estimating the mean and variance of the measured round-trip time (RTT), i.e., the time interval between sending a segment and receiving an acknowledgment for it [Jacobson88a].

Section 4.2 introduces a new TCP option, "Timestamps", and then defines a mechanism using this option that allows nearly every segment, including retransmissions, to be timed at negligible computational cost. We use the mnemonic RTTM (Round Trip Time Measurement) for this mechanism, to distinguish it from other uses of the Timestamps option.

1.2. TCP Reliability

An especially serious kind of error may result from an accidental reuse of TCP sequence numbers in data segments. TCP reliability depends upon the existence of a bound on the lifetime of a segment: the "Maximum Segment Lifetime" or MSL.

Duplication of sequence numbers might happen in either of two ways:

(1) Sequence number wrap-around on the current connection

A TCP sequence number contains 32 bits. At a high enough transfer rate, the 32-bit sequence space may be "wrapped" (cycled) within the time that a segment is delayed in queues.

(2) Earlier incarnation of the connection

Suppose that a connection terminates, either by a proper close sequence or due to a host crash, and the same connection (i.e., using the same pair of port numbers) is immediately reopened. A delayed segment from the terminated connection could fall within

the current window for the new incarnation and be accepted as valid.

Duplicates from earlier incarnations, Case (2), are avoided by enforcing the current fixed MSL of the TCP spec, as explained in Section 5.3 and Appendix B. However, case (1), avoiding the reuse of sequence numbers within the same connection, requires an MSL bound that depends upon the transfer rate, and at high enough rates, a new mechanism is required.

A possible fix for the problem of cycling the sequence space would be to increase the size of the TCP sequence number field. For example, the sequence number field (and also the acknowledgment field) could be expanded to 64 bits. This could be done either by changing the TCP header or by means of an additional option.

Section 5 presents a different mechanism, which we call PAWS (Protection Against Wrapped Sequence numbers), to extend TCP reliability to transfer rates well beyond the foreseeable upper limit of network bandwidths. PAWS uses the TCP Timestamps option defined in Section 4.2 to protect against old duplicates from the same connection.

1.3. Using TCP options

The extensions defined in this document all use new TCP options.

When RFC 1323 was published, there was concern that some buggy TCP implementation might be crashed by the first appearance of an option on a non-SYN segment. However, bugs like that can lead to DOS attacks against a TCP, so it is now expected that most TCP implementations will properly handle unknown options on non-SYN segments. But it is still prudent to be conservative in what you send, and avoiding buggy TCP implementation is not the only reason for negotiating TCP options on SYN segments. Therefore, for each of the extensions defined below, it is recommended that TCP options will be sent on non-SYN segments only after an exchange of options on the SYN segments has indicated that both sides understand the extension. Furthermore, an extension option will be sent in a <SYN,ACK> segment only if the corresponding option was received in the initial <SYN> segment.

The timestamps option may appear in any data or ACK segment, adding 12 bytes to the 20-byte TCP header. We believe that the bandwidth saved by reducing unnecessary retransmission timeouts will more than pay for the extra header bandwidth.

Appendix A contains a recommended layout of the options in TCP

headers to achieve reasonable data field alignment.

Finally, we observe that most of the mechanisms defined in this memo are important for LFN's and/or very high-speed networks. For low-speed networks, it might be a performance optimization to NOT use these mechanisms. A TCP vendor concerned about optimal performance over low-speed paths might consider turning these extensions off for low-speed paths, or allow a user or installation manager to disable them.

2. 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].

3. TCP Window Scale Option

3.1. Introduction

The window scale extension expands the definition of the TCP window to 32 bits and then uses a scale factor to carry this 32-bit value in the 16-bit Window field of the TCP header (SEG.WND in RFC 793). The scale factor is carried in a new TCP option, Window Scale. This option is sent only in a SYN segment (a segment with the SYN bit on), hence the window scale is fixed in each direction when a connection is opened.

The maximum receive window, and therefore the scale factor, is determined by the maximum receive buffer space. In a typical modern implementation, this maximum buffer space is set by default but can be overridden by a user program before a TCP connection is opened. This determines the scale factor, and therefore no new user interface is needed for window scaling.

3.2. Window Scale Option

The three-byte Window Scale option MAY be sent in a SYN segment by a TCP. It has two purposes: (1) indicate that the TCP is prepared to do both send and receive window scaling, and (2) communicate a scale factor to be applied to its receive window. Thus, a TCP that is prepared to scale windows SHOULD send the option, even if its own scale factor is 1. The scale factor is limited to a power of two and encoded logarithmically, so it may be implemented by binary shift operations.

TCP Window Scale Option (WSopt):

Kind: 3

Length: 3 bytes

```
+---------+---------+---------+
| Kind=3  |Length=3 |shift.cnt|
+---------+---------+---------+
     1         1         1
```

This option is an offer, not a promise; both sides MUST send Window Scale options in their SYN segments to enable window scaling in either direction. If window scaling is enabled, then the TCP that sent this option will right-shift its true receive-window values by 'shift.cnt' bits for transmission in SEG.WND. The value 'shift.cnt' MAY be zero (offering to scale, while applying a scale factor of 1 to the receive window).

This option MAY be sent in an initial <SYN> segment (i.e., a segment with the SYN bit on and the ACK bit off). It MAY also be sent in a <SYN,ACK> segment, but only if a Window Scale option was received in the initial <SYN> segment. A Window Scale option in a segment without a SYN bit SHOULD be ignored.

The Window field in a SYN (i.e., a <SYN> or <SYN,ACK>) segment itself is never scaled.

3.3. Using the Window Scale Option

A model implementation of window scaling is as follows, using the notation of [RFC0793]:

- o All windows are treated as 32-bit quantities for storage in the connection control block and for local calculations. This includes the send-window (SND.WND) and the receive-window (RCV.WND) values, as well as the congestion window.

- o The connection state is augmented by two window shift counts, Snd.Wind.Scale and Rcv.Wind.Scale, to be applied to the incoming and outgoing window fields, respectively.

- o If a TCP receives a <SYN> segment containing a Window Scale option, it sends its own Window Scale option in the <SYN,ACK> segment.

- o The Window Scale option is sent with shift.cnt = R, where R is the value that the TCP would like to use for its receive window.

- o Upon receiving a SYN segment with a Window Scale option containing shift.cnt = S, a TCP sets Snd.Wind.Scale to S and sets Rcv.Wind.Scale to R; otherwise, it sets both Snd.Wind.Scale and Rcv.Wind.Scale to zero.

- o The window field (SEG.WND) in the header of every incoming segment, with the exception of SYN segments, is left-shifted by Snd.Wind.Scale bits before updating SND.WND:

  SND.WND = SEG.WND << Snd.Wind.Scale

  (assuming the other conditions of [RFC0793] are met, and using the "C" notation "<<" for left-shift).

- o The window field (SEG.WND) of every outgoing segment, with the exception of SYN segments, is right-shifted by Rcv.Wind.Scale bits:

  SND.WND = RCV.WND >> Rcv.Wind.Scale

TCP determines if a data segment is "old" or "new" by testing whether its sequence number is within 2^31 bytes of the left edge of the window, and if it is not, discarding the data as "old". To insure that new data is never mistakenly considered old and vice versa, the left edge of the sender’s window has to be at most 2^31 away from the right edge of the receiver’s window. Similarly with the sender’s right edge and receiver’s left edge. Since the right and left edges of either the sender’s or receiver’s window differ by the window size, and since the sender and receiver windows can be out of phase by at most the window size, the above constraints imply that two times the max window size must be less than 2^31, or

max window < 2^30

Since the max window is 2^S (where S is the scaling shift count) times at most 2^16 - 1 (the maximum unscaled window), the maximum window is guaranteed to be < 2^30 if S <= 14. Thus, the shift count MUST be limited to 14 (which allows windows of 2^30 = 1 Gbyte). If a Window Scale option is received with a shift.cnt value exceeding 14, the TCP SHOULD log the error but use 14 instead of the specified value.

The scale factor applies only to the Window field as transmitted in the TCP header; each TCP using extended windows will maintain the window values locally as 32-bit numbers. For example, the "congestion window" computed by Slow Start and Congestion Avoidance is not affected by the scale factor, so window scaling will not introduce quantization into the congestion window.

3.4. Addressing Window Retraction

When a non-zero scale factor is in use, there are instances when a retracted window can be offered [Mathis08]. The end of the window will be on a boundary based on the granularity of the scale factor being used. If the sequence number is then updated by a number of bytes smaller than that granularity, the TCP will have to either advertise a new window that is beyond what it previously advertised (and perhaps beyond the buffer), or will have to advertise a smaller window, which will cause the TCP window to shrink. Implementations MUST ensure that they handle a shrinking window, as specified in section 4.2.2.16 of [RFC1122].

For the receiver, this implies that:

1) The receiver MUST honor, as in-window, any segment that would have been in-window for any ACK sent by the receiver.

2) When window scaling is in effect, the receiver SHOULD track the actual maximum window sequence number (which is likely to be greater than the window announced by the most recent ACK, if more than one segment has arrived since the application consumed any data in the receive buffer).

On the sender side:

3) The initial transmission MUST honor window on most recent ACK.

4) On first retransmission, or if the sequence number is out-of-window by less than (2^Rcv.Wind.Scale) then do normal retransmission(s) without regard to receiver window as long as the original segment was in window when it was sent.

5) On subsequent retransmissions, treat such ACKs as zero window probes.

4. RTTM -- Round-Trip Time Measurement

4.1. Introduction

Accurate and current RTT estimates are necessary to adapt to changing traffic conditions and to avoid an instability known as "congestion collapse" [RFC0896] in a busy network. However, accurate measurement of RTT may be difficult both in theory and in implementation.

Many TCP implementations base their RTT measurements upon a sample of one packet per window or less. While this yields an adequate approximation to the RTT for small windows, it results in an unacceptably poor RTT estimate for a LFN. If we look at RTT estimation as a signal processing problem (which it is), a data signal at some frequency, the packet rate, is being sampled at a lower frequency, the window rate. This lower sampling frequency violates Nyquist's criteria and may therefore introduce "aliasing" artifacts into the estimated RTT [Hamming77].

A good RTT estimator with a conservative retransmission timeout calculation can tolerate aliasing when the sampling frequency is "close" to the data frequency. For example, with a window of 8 packets, the sample rate is 1/8 the data frequency -- less than an order of magnitude different. However, when the window is tens or hundreds of packets, the RTT estimator may be seriously in error, resulting in spurious retransmissions.

If there are dropped packets, the problem becomes worse. Zhang [Zhang86], Jain [Jain86] and Karn [Karn87] have shown that it is not possible to accumulate reliable RTT estimates if retransmitted segments are included in the estimate. Since a full window of data will have been transmitted prior to a retransmission, all of the segments in that window will have to be ACKed before the next RTT sample can be taken. This means at least an additional window's worth of time between RTT measurements and, as the error rate approaches one per window of data (e.g., 10^-6 errors per bit for the Wideband satellite network), it becomes effectively impossible to obtain a valid RTT measurement.

A solution to these problems, which actually simplifies the sender substantially, is as follows: using TCP options, the sender places a timestamp in each data segment, and the receiver reflects these timestamps back in ACK segments. Then a single subtract gives the sender an accurate RTT measurement for every ACK segment (which will correspond to every other data segment, with a sensible receiver). We call this the RTTM (Round-Trip Time Measurement) mechanism.

It is vitally important to use the RTTM mechanism with big windows;

otherwise, the door is opened to some dangerous instabilities due to aliasing. Furthermore, the option is probably useful for all TCP's, since it simplifies the sender.

4.2. TCP Timestamps Option

TCP is a symmetric protocol, allowing data to be sent at any time in either direction, and therefore timestamp echoing may occur in either direction. For simplicity and symmetry, we specify that timestamps always be sent and echoed in both directions. For efficiency, we combine the timestamp and timestamp reply fields into a single TCP Timestamps Option.

TCP Timestamps Option (TSopt):

Kind: 8

Length: 10 bytes

```
+-------+-------+---------------------+---------------------+
|Kind=8 |  10   |   TS Value (TSval)  |TS Echo Reply (TSecr)|
+-------+-------+---------------------+---------------------+
    1       1              4                     4
```

The Timestamps option carries two four-byte timestamp fields. The Timestamp Value field (TSval) contains the current value of the timestamp clock of the TCP sending the option.

The Timestamp Echo Reply field (TSecr) is valid if the ACK bit is set in the TCP header; if it is valid, it echoes a timestamp value that was sent by the remote TCP in the TSval field of a Timestamp option. When TSecr is not valid, its value MUST be zero. However, a value of zero does not imply TSecr being invalid. The TSecr value will generally be from the most recent Timestamps Option that was received; however, there are exceptions that are explained below.

A TCP MAY send the Timestamps option (TSopt) in an initial <SYN> segment (i.e., a segment containing a SYN bit and no ACK bit). Once a TSopt has been sent or received in a non <SYN> segment, it MUST be sent in all segments. Once a TSopt has been received in a non <SYN> segment, then any successive segment that is received without the RST bit and without a TSopt MAY be dropped without further processing, and an ACK of the current SND.UNA generated.

In the case of crossing SYN packets where one SYN contains a TSopt and the other doesn't, both sides SHOULD put a TSopt in the <SYN,ACK> segment.

4.3. The RTTM Mechanism

RTTM places a Timestamps option in every segment, with a TSval that is obtained from a (virtual) "timestamp clock". Values of this clock MUST be at least approximately proportional to real time, in order to measure actual RTT.

These TSval values are echoed in TSecr values in the reverse direction. The difference between a received TSecr value and the current timestamp clock value provides a RTT measurement.

When timestamps are used, every segment that is received will contain a TSecr value. However, these values cannot all be used to update the measured RTT. The following example illustrates why. It shows a one-way data flow with segments arriving in sequence without loss. Here A, B, C... represent data blocks occupying successive blocks of sequence numbers, and ACK(A),... represent the corresponding cumulative acknowledgments. The two timestamp fields of the Timestamps option are shown symbolically as <TSval=x,TSecr=y>. Each TSecr field contains the value most recently received in a TSval field.

```
TCP  A                                        TCP B

                 <A,TSval=1,TSecr=120> ----->

     <---- <ACK(A),TSval=127,TSecr=1>

                 <B,TSval=5,TSecr=127> ----->

     <---- <ACK(B),TSval=131,TSecr=5>

   . . . . . . . . . . . . . . . . . . . . . .

                 <C,TSval=65,TSecr=131> ---->

     <---- <ACK(C),TSval=191,TSecr=65>

                      (etc.)
```

The dotted line marks a pause (60 time units long) in which A had nothing to send. Note that this pause inflates the RTT which B could infer from receiving TSecr=131 in data segment C. Thus, in one-way data flows, RTTM in the reverse direction measures a value that is inflated by gaps in sending data. However, the following rule prevents a resulting inflation of the measured RTT:

RTTM Rule: A TSecr value received in a segment is used to update the averaged RTT measurement only if

a) the segment acknowledges some new data, i.e., only if it advances the left edge of the send window, and

b) the segment does not indicate any loss or reordering, i.e. contains SACK options

Since TCP B is not sending data, the data segment C does not acknowledge any new data when it arrives at B. Thus, the inflated RTTM measurement is not used to update B's RTTM measurement.

Implementers should note that with Timestamps multiple RTTMs can be taken per RTT. Many RTO estimators have a weighting factor based on an implicit assumption that at most one RTTM will be sampled per RTT. When using multiple RTTMs per RTT to update the RTO estimator, the weighting factor needs to be decreased to take into account the more frequent RTTMs. For example, an implementation could choose to just use one sample per RTT to update the RTO estimator, or vary the gain based on the congestion window, or take an average of all the RTTM measurements received over one RTT, and then use that value to update the RTO estimator. This document does not prescribe any particular method for modifying the RTO estimator.

4.4. Which Timestamp to Echo

If more than one Timestamps option is received before a reply segment is sent, the TCP must choose only one of the TSvals to echo, ignoring the others. To minimize the state kept in the receiver (i.e., the number of unprocessed TSvals), the receiver should be required to retain at most one timestamp in the connection control block.

There are three situations to consider:

(A) Delayed ACKs.

Many TCP's acknowledge only every Kth segment out of a group of segments arriving within a short time interval; this policy is known generally as "delayed ACKs". The data-sender TCP must measure the effective RTT, including the additional time due to delayed ACKs, or else it will retransmit unnecessarily. Thus, when delayed ACKs are in use, the receiver SHOULD reply with the TSval field from the earliest unacknowledged segment.

(B) A hole in the sequence space (segment(s) have been lost).

The sender will continue sending until the window is filled, and the receiver may be generating ACKs as these out-of-order segments arrive (e.g., to aid "fast retransmit").

The lost segment is probably a sign of congestion, and in that situation the sender should be conservative about retransmission. Furthermore, it is better to overestimate than underestimate the RTT. An ACK for an out-of-order segment SHOULD therefore contain the timestamp from the most recent segment that advanced the window.

The same situation occurs if segments are re-ordered by the network.

(C) A filled hole in the sequence space.

The segment that fills the hole represents the most recent measurement of the network characteristics. A RTT computed from an earlier segment would probably include the sender's retransmit time-out, badly biasing the sender's average RTT estimate. Thus, the timestamp from the latest segment (which filled the hole) MUST be echoed.

An algorithm that covers all three cases is described in the following rules for Timestamps option processing on a synchronized connection:

(1) The connection state is augmented with two 32-bit slots:

TS.Recent holds a timestamp to be echoed in TSecr whenever a segment is sent, and Last.ACK.sent holds the ACK field from the last segment sent. Last.ACK.sent will equal RCV.NXT except when ACKs have been delayed.

(2) If:

```
SEG.TSval >= TS.recent and SEG.SEQ <= Last.ACK.sent
```

then SEG.TSval is copied to TS.Recent; otherwise, it is ignored.

(3) When a TSopt is sent, its TSecr field is set to the current TS.Recent value.

The following examples illustrate these rules. Here A, B, C... represent data segments occupying successive blocks of sequence numbers, and ACK(A),... represent the corresponding acknowledgment

segments. Note that ACK(A) has the same sequence number as B. We show only one direction of timestamp echoing, for clarity.

o Packets arrive in sequence, and some of the ACKs are delayed.

By case (A), the timestamp from the oldest unacknowledged segment is echoed.

```
                                          TS.Recent
            <A, TSval=1> -------------------->
                                                1
            <B, TSval=2> -------------------->
                                                1
            <C, TSval=3> -------------------->
                                                1
                     <---- <ACK(C), TSecr=1>
            (etc)
```

o Packets arrive out of order, and every packet is acknowledged.

By case (B), the timestamp from the last segment that advanced the left window edge is echoed, until the missing segment arrives; it is echoed according to Case (C). The same sequence would occur if segments B and D were lost and retransmitted.

```
                                          TS.Recent
            <A, TSval=1> -------------------->
                                                1
                     <---- <ACK(A), TSecr=1>
                                                1
            <C, TSval=3> -------------------->
                                                1
                     <---- <ACK(A), TSecr=1>
                                                1
            <B, TSval=2> -------------------->
                                                2
                     <---- <ACK(C), TSecr=2>
                                                2
            <E, TSval=5> -------------------->
                                                2
                     <---- <ACK(C), TSecr=2>
                                                2
            <D, TSval=4> -------------------->
                                                4
                     <---- <ACK(E), TSecr=4>
            (etc)
```

5. PAWS -- Protection Against Wrapped Sequence Numbers

5.1. Introduction

Section 5.2 describes a simple mechanism to reject old duplicate segments that might corrupt an open TCP connection; we call this mechanism PAWS (Protection Against Wrapped Sequence numbers). PAWS operates within a single TCP connection, using state that is saved in the connection control block. Section 5.3 and Appendix G discuss the implications of the PAWS mechanism for avoiding old duplicates from previous incarnations of the same connection.

5.2. The PAWS Mechanism

PAWS uses the same TCP Timestamps option as the RTTM mechanism described earlier, and assumes that every received TCP segment (including data and ACK segments) contains a timestamp SEG.TSval whose values are monotonically non-decreasing in time. The basic idea is that a segment can be discarded as an old duplicate if it is received with a timestamp SEG.TSval less than some timestamp recently received on this connection.

In both the PAWS and the RTTM mechanism, the "timestamps" are 32-bit unsigned integers in a modular 32-bit space. Thus, "less than" is defined the same way it is for TCP sequence numbers, and the same implementation techniques apply. If s and t are timestamp values,

$$s < t \quad \text{if } 0 < (t - s) < 2^{31},$$

computed in unsigned 32-bit arithmetic.

The choice of incoming timestamps to be saved for this comparison MUST guarantee a value that is monotonically increasing. For example, we might save the timestamp from the segment that last advanced the left edge of the receive window, i.e., the most recent in-sequence segment. Instead, we choose the value TS.Recent introduced in Section 4.4 for the RTTM mechanism, since using a common value for both PAWS and RTTM simplifies the implementation of both. As Section 4.4 explained, TS.Recent differs from the timestamp from the last in-sequence segment only in the case of delayed ACKs, and therefore by less than one window. Either choice will therefore protect against sequence number wrap-around.

RTTM was specified in a symmetrical manner, so that TSval timestamps are carried in both data and ACK segments and are echoed in TSecr fields carried in returning ACK or data segments. PAWS submits all incoming segments to the same test, and therefore protects against duplicate ACK segments as well as data segments. (An alternative

non-symmetric algorithm would protect against old duplicate ACKs: the sender of data would reject incoming ACK segments whose TSecr values were less than the TSecr saved from the last segment whose ACK field advanced the left edge of the send window. This algorithm was deemed to lack economy of mechanism and symmetry.)

TSval timestamps sent on <SYN> and <SYN,ACK> segments are used to initialize PAWS. PAWS protects against old duplicate non-SYN segments, and duplicate SYN segments received while there is a synchronized connection. Duplicate <SYN> and <SYN,ACK> segments received when there is no connection will be discarded by the normal 3-way handshake and sequence number checks of TCP.

[RFC1323] recommended that RST segments NOT carry timestamps, and that they be acceptable regardless of their timestamp. At that time, the thinking was that old duplicate RST segments should be exceedingly unlikely, and their cleanup function should take precedence over timestamps. More recently, discussions about various blind attacks on TCP connections have raised the suggestion that if the Timestamps option is present, SEG.TSecr could be used to provide stricter acceptance tests for RST packets. While still under discussion, to enable research into this area it is now RECOMMENDED that when generating a RST, that if the packet causing the RST to be generated contained a Timestamps option that the RST also contain a Timestamps option. In the RST segment, SEG.TSecr SHOULD be set to SEG.TSval from the incoming packet and SEG.TSval SHOULD be set to zero. If a RST is being generated because of a user abort, and Snd.TS.OK is set, then a Timestamps option SHOULD be included in the RST. When a RST packet is received, it MUST NOT be subjected to PAWS checks, and information from the Timestamps option MUST NOT be used to update connection state information. SEG.TSecr MAY be used to provide stricter RST acceptance checks.

5.2.1. Basic PAWS Algorithm

The PAWS algorithm requires the following processing to be performed on all incoming segments for a synchronized connection:

R1) If there is a Timestamps option in the arriving segment, SEG.TSval < TS.Recent, TS.Recent is valid (see later discussion) and the RST bit is not set, then treat the arriving segment as not acceptable:

> Send an acknowledgement in reply as specified in [RFC0793] page 69 and drop the segment.
>
> Note: it is necessary to send an ACK segment in order to retain TCP's mechanisms for detecting and recovering from

half-open connections. For example, see Figure 10 of [RFC0793].

R2) If the segment is outside the window, reject it (normal TCP processing)

R3) If an arriving segment satisfies: SEG.SEQ <= Last.ACK.sent (see Section 4.4), then record its timestamp in TS.Recent.

R4) If an arriving segment is in-sequence (i.e., at the left window edge), then accept it normally.

R5) Otherwise, treat the segment as a normal in-window, out-of-sequence TCP segment (e.g., queue it for later delivery to the user).

Steps R2, R4, and R5 are the normal TCP processing steps specified by [RFC0793].

It is important to note that the timestamp is checked only when a segment first arrives at the receiver, regardless of whether it is in-sequence or it must be queued for later delivery.

Consider the following example.

Suppose the segment sequence: A.1, B.1, C.1, ..., Z.1 has been sent, where the letter indicates the sequence number and the digit represents the timestamp. Suppose also that segment B.1 has been lost. The timestamp in TS.Recent is 1 (from A.1), so C.1, ..., Z.1 are considered acceptable and are queued. When B is retransmitted as segment B.2 (using the latest timestamp), it fills the hole and causes all the segments through Z to be acknowledged and passed to the user. The timestamps of the queued segments are *not* inspected again at this time, since they have already been accepted. When B.2 is accepted, TS.Recent is set to 2.

This rule allows reasonable performance under loss. A full window of data is in transit at all times, and after a loss a full window less one packet will show up out-of-sequence to be queued at the receiver (e.g., up to ~2^30 bytes of data); the timestamp option must not result in discarding this data.

In certain unlikely circumstances, the algorithm of rules R1-R5 could lead to discarding some segments unnecessarily, as shown in the following example:

Suppose again that segments: A.1, B.1, C.1, ..., Z.1 have been sent in sequence and that segment B.1 has been lost. Furthermore, suppose delivery of some of C.1, ... Z.1 is delayed until AFTER the retransmission B.2 arrives at the receiver. These delayed segments will be discarded unnecessarily when they do arrive, since their timestamps are now out of date.

This case is very unlikely to occur. If the retransmission was triggered by a timeout, some of the segments C.1, ... Z.1 must have been delayed longer than the RTO time. This is presumably an unlikely event, or there would be many spurious timeouts and retransmissions. If B's retransmission was triggered by the "fast retransmit" algorithm, i.e., by duplicate ACKs, then the queued segments that caused these ACKs must have been received already.

Even if a segment were delayed past the RTO, the Fast Retransmit mechanism [Jacobson90c] will cause the delayed packets to be retransmitted at the same time as B.2, avoiding an extra RTT and therefore causing a very small performance penalty.

We know of no case with a significant probability of occurrence in which timestamps will cause performance degradation by unnecessarily discarding segments.

5.2.2. Timestamp Clock

It is important to understand that the PAWS algorithm does not require clock synchronization between sender and receiver. The sender's timestamp clock is used to stamp the segments, and the sender uses the echoed timestamp to measure RTTs. However, the receiver treats the timestamp as simply a monotonically increasing serial number, without any necessary connection to its clock. From the receiver's viewpoint, the timestamp is acting as a logical extension of the high-order bits of the sequence number.

The receiver algorithm does place some requirements on the frequency of the timestamp clock.

(a) The timestamp clock must not be "too slow".

It MUST tick at least once for each 2^31 bytes sent. In fact, in order to be useful to the sender for round trip timing, the clock SHOULD tick at least once per window's worth of data, and even with the window extension defined in Section 3.2, 2^31 bytes must be at least two windows.

To make this more quantitative, any clock faster than 1 tick/sec will reject old duplicate segments for link speeds of ~8 Gbps.

A 1 ms timestamp clock will work at link speeds up to 8 Tbps (8*10^12) bps!

(b) The timestamp clock must not be "too fast".

The recycling time of the timestamp clock MUST be greater than MSL seconds. Since the clock (timestamp) is 32 bits and the worst-case MSL is 255 seconds, the maximum acceptable clock frequency is one tick every 59 ns.

However, it is desirable to establish a much longer recycle period, in order to handle outdated timestamps on idle connections (see Section 5.2.3), and to relax the MSL requirement for preventing sequence number wrap-around. With a 1 ms timestamp clock, the 32-bit timestamp will wrap its sign bit in 24.8 days. Thus, it will reject old duplicates on the same connection if MSL is 24.8 days or less. This appears to be a very safe figure; an MSL of 24.8 days or longer can probably be assumed in the internet without requiring precise MSL enforcement.

Based upon these considerations, we choose a timestamp clock frequency in the range 1 ms to 1 sec per tick. This range also matches the requirements of the RTTM mechanism, which does not need much more resolution than the granularity of the retransmit timer, e.g., tens or hundreds of milliseconds.

The PAWS mechanism also puts a strong monotonicity requirement on the sender’s timestamp clock. The method of implementation of the timestamp clock to meet this requirement depends upon the system hardware and software.

- Some hosts have a hardware clock that is guaranteed to be monotonic between hardware resets.

- A clock interrupt may be used to simply increment a binary integer by 1 periodically.

- The timestamp clock may be derived from a system clock that is subject to being abruptly changed, by adding a variable offset value. This offset is initialized to zero. When a new timestamp clock value is needed, the offset can be adjusted as necessary to make the new value equal to or larger than the previous value (which was saved for this purpose).

5.2.3. Outdated Timestamps

If a connection remains idle long enough for the timestamp clock of the other TCP to wrap its sign bit, then the value saved in TS.Recent will become too old; as a result, the PAWS mechanism will cause all subsequent segments to be rejected, freezing the connection (until the timestamp clock wraps its sign bit again).

With the chosen range of timestamp clock frequencies (1 sec to 1 ms), the time to wrap the sign bit will be between 24.8 days and 24800 days. A TCP connection that is idle for more than 24 days and then comes to life is exceedingly unusual. However, it is undesirable in principle to place any limitation on TCP connection lifetimes.

We therefore require that an implementation of PAWS include a mechanism to "invalidate" the TS.Recent value when a connection is idle for more than 24 days. (An alternative solution to the problem of outdated timestamps would be to send keep-alive segments at a very low rate, but still more often than the wrap-around time for timestamps, e.g., once a day. This would impose negligible overhead. However, the TCP specification has never included keep-alives, so the solution based upon invalidation was chosen.)

Note that a TCP does not know the frequency, and therefore, the wraparound time, of the other TCP, so it must assume the worst. The validity of TS.Recent needs to be checked only if the basic PAWS timestamp check fails, i.e., only if SEG.TSval < TS.Recent. If TS.Recent is found to be invalid, then the segment is accepted, regardless of the failure of the timestamp check, and rule R3 updates TS.Recent with the TSval from the new segment.

To detect how long the connection has been idle, the TCP MAY update a clock or timestamp value associated with the connection whenever TS.Recent is updated, for example. The details will be implementation-dependent.

5.2.4. Header Prediction

"Header prediction" [Jacobson90a] is a high-performance transport protocol implementation technique that is most important for high-speed links. This technique optimizes the code for the most common case, receiving a segment correctly and in order. Using header prediction, the receiver asks the question, "Is this segment the next in sequence?" This question can be answered in fewer machine instructions than the question, "Is this segment within the window?"

Adding header prediction to our timestamp procedure leads to the following recommended sequence for processing an arriving TCP

segment:

H1) Check timestamp (same as step R1 above)

H2) Do header prediction: if segment is next in sequence and if there are no special conditions requiring additional processing, accept the segment, record its timestamp, and skip H3.

H3) Process the segment normally, as specified in RFC 793. This includes dropping segments that are outside the window and possibly sending acknowledgments, and queuing in-window, out-of-sequence segments.

Another possibility would be to interchange steps H1 and H2, i.e., to perform the header prediction step H2 FIRST, and perform H1 and H3 only when header prediction fails. This could be a performance improvement, since the timestamp check in step H1 is very unlikely to fail, and it requires unsigned modulo arithmetic. To perform this check on every single segment is contrary to the philosophy of header prediction. We believe that this change might produce a measurable reduction in CPU time for TCP protocol processing on high-speed networks.

However, putting H2 first would create a hazard: a segment from 2^32 bytes in the past might arrive at exactly the wrong time and be accepted mistakenly by the header-prediction step. The following reasoning has been introduced in [RFC1185] to show that the probability of this failure is negligible.

> If all segments are equally likely to show up as old duplicates, then the probability of an old duplicate exactly matching the left window edge is the maximum segment size (MSS) divided by the size of the sequence space. This ratio must be less than 2^-16, since MSS must be < 2^16; for example, it will be (2^12)/(2^32) = 2^-20 for a FDDI link. However, the older a segment is, the less likely it is to be retained in the Internet, and under any reasonable model of segment lifetime the probability of an old duplicate exactly at the left window edge must be much smaller than 2^-16.
>
> The 16 bit TCP checksum also allows a basic unreliability of one part in 2^16. A protocol mechanism whose reliability exceeds the reliability of the TCP checksum should be considered "good enough", i.e., it won’t contribute significantly to the overall error rate. We therefore believe we can ignore the problem of an old duplicate being accepted by doing header prediction before checking the timestamp.

However, this probabilistic argument is not universally accepted, and

the consensus at present is that the performance gain does not justify the hazard in the general case. It is therefore recommended that H2 follow H1.

5.2.5. IP Fragmentation

At high data rates, the protection against old packets provided by PAWS can be circumvented by errors in IP fragment reassembly (see [RFC4963]). The only way to protect against incorrect IP fragment reassembly is to not allow the packets to be fragmented. This is done by setting the Don't Fragment (DF) bit in the IP header. Setting the DF bit implies the use of Path MTU Discovery as described in [RFC1191], [RFC1981], and [RFC4821], thus any TCP implementation that implements PAWS MUST also implement Path MTU Discovery.

5.3. Duplicates from Earlier Incarnations of Connection

The PAWS mechanism protects against errors due to sequence number wrap-around on high-speed connections. Segments from an earlier incarnation of the same connection are also a potential cause of old duplicate errors. In both cases, the TCP mechanisms to prevent such errors depend upon the enforcement of a maximum segment lifetime (MSL) by the Internet (IP) layer (see Appendix of RFC 1185 for a detailed discussion). Unlike the case of sequence space wrap-around, the MSL required to prevent old duplicate errors from earlier incarnations does not depend upon the transfer rate. If the IP layer enforces the recommended 2 minute MSL of TCP, and if the TCP rules are followed, TCP connections will be safe from earlier incarnations, no matter how high the network speed. Thus, the PAWS mechanism is not required for this case.

We may still ask whether the PAWS mechanism can provide additional security against old duplicates from earlier connections, allowing us to relax the enforcement of MSL by the IP layer. Appendix B explores this question, showing that further assumptions and/or mechanisms are required, beyond those of PAWS. This is not part of the current extension.

6. Conclusions and Acknowledgements

This memo presented a set of extensions to TCP to provide efficient operation over large-bandwidth*delay-product paths and reliable operation over very high-speed paths. These extensions are designed to provide compatible interworking with TCP's that do not implement the extensions.

These mechanisms are implemented using new TCP options for scaled

windows and timestamps. The timestamps are used for two distinct mechanisms: RTTM (Round Trip Time Measurement) and PAWS (Protection Against Wrapped Sequences).

The Window Scale option was originally suggested by Mike St. Johns of USAF/DCA. The present form of the option was suggested by Mike Karels of UC Berkeley in response to a more cumbersome scheme defined by Van Jacobson. Lixia Zhang helped formulate the PAWS mechanism description in [RFC1185].

Finally, much of this work originated as the result of discussions within the End-to-End Task Force on the theoretical limitations of transport protocols in general and TCP in particular. Task force members and other on the end2end-interest list have made valuable contributions by pointing out flaws in the algorithms and the documentation. Continued discussion and development since the publication of [RFC1323] originally occurred in the IETF TCP Large Windows Working Group, later on in the End-to-End Task Force, and most recently in the IETF TCP Maintenance Working Group. The authors are grateful for all these contributions.

7. Security Considerations

The TCP sequence space is a fixed size, and as the window becomes larger it becomes easier for an attacker to generate forged packets that can fall within the TCP window, and be accepted as valid packets. While use of Timestamps and PAWS can help to mitigate this, when using PAWS, if an attacker is able to forge a packet that is acceptable to the TCP connection, a timestamp that is in the future would cause valid packets to be dropped due to PAWS checks. Hence, implementers should take care to not open the TCP window drastically beyond the requirements of the connection.

Middle boxes and options: If a middle box removes TCP options from the SYN, such as TSopt, a high speed connection that needs PAWS would not have that protection. In this situation, an implementer could provide a mechanism for the application to determine whether or not PAWS is in use on the connection, and chose to terminate the connection if that protection doesn’t exist.

Mechanisms to protect the TCP header from modification should also protect the TCP options.

A naive implementation that derives the timestamp clock value directly from a system uptime clock may unintentionally leak this information to an attacker. This does not directly compromise any of the mechanisms described in this document. However, this may be

valuable information to a potential attacker. An implementer should evaluate the potential impact and mitigate this accordingly (i.e. by using a random offset for the timestamp clock on each connection, or using an external, real-time derived timestamp clock source).

Expanding the TCP window beyond 64K for IPv6 allows Jumbograms [RFC2675] to be used when the local network supports packets larger than 64K. When larger TCP packets are used, the TCP checksum becomes weaker.

8. IANA Considerations

This document has no actions for IANA.

9. References

9.1. Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[RFC1191] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, November 1990.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

9.2. Informative References

[Garlick77]
Garlick, L., Rom, R., and J. Postel, "Issues in Reliable Host-to-Host Protocols", Proc. Second Berkeley Workshop on Distributed Data Management and Computer Networks, May 1977, <http://www.rfc-editor.org/ien/ien12.txt>.

[Hamming77]
Hamming, R., "Digital Filters", Prentice Hall, Englewood Cliffs, N.J. ISBN 0-13-212571-4, 1977.

[Jacobson88a]
Jacobson, V., "Congestion Avoidance and Control", SIGCOMM '88, Stanford, CA., August 1988, <http://ee.lbl.gov/papers/congavoid.pdf>.

[Jacobson90a]
Jacobson, V., "4BSD Header Prediction", ACM Computer

Communication Review, April 1990.

[Jacobson90c]
Jacobson, V., "Modified TCP congestion avoidance algorithm", Message to the end2end-interest mailing list, April 1990, <ftp://ftp.isi.edu/end2end/end2end-interest-1990.mail>.

[Jain86] Jain, R., "Divergence of Timeout Algorithms for Packet Retransmissions", Proc. Fifth Phoenix Conf. on Comp. and Comm., Scottsdale, Arizona, March 1986, <http://arxiv.org/ftp/cs/papers/9809/9809097.pdf>.

[Karn87] Karn, P. and C. Partridge, "Estimating Round-Trip Times in Reliable Transport Protocols", Proc. SIGCOMM '87, August 1987.

[Martin03]
Martin, D., "[Tsvwg] RFC 1323.bis", Message to the tsvwg mailing list, September 2003, <http://www.ietf.org/mail-archive/web/tsvwg/current/msg04435.html>.

[Mathis08]
Mathis, M., "[tcpm] Example of 1323 window retraction problem", Message to the tcpm mailing list, March 2008, <http://www.ietf.org/mail-archive/web/tcpm/current/msg03564.html>.

[RFC0896] Nagle, J., "Congestion control in IP/TCP internetworks", RFC 896, January 1984.

[RFC1072] Jacobson, V. and R. Braden, "TCP extensions for long-delay paths", RFC 1072, October 1988.

[RFC1110] McKenzie, A., "Problem with the TCP big window option", RFC 1110, August 1989.

[RFC1122] Braden, R., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, October 1989.

[RFC1185] Jacobson, V., Braden, B., and L. Zhang, "TCP Extension for High-Speed Paths", RFC 1185, October 1990.

[RFC1323] Jacobson, V., Braden, B., and D. Borman, "TCP Extensions for High Performance", RFC 1323, May 1992.

[RFC1981] McCann, J., Deering, S., and J. Mogul, "Path MTU Discovery for IP version 6", RFC 1981, August 1996.

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

[RFC2581] Allman, M., Paxson, V., and W. Stevens, "TCP Congestion Control", RFC 2581, April 1999.

[RFC2675] Borman, D., Deering, S., and R. Hinden, "IPv6 Jumbograms", RFC 2675, August 1999.

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

[RFC4821] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, March 2007.

[RFC4963] Heffner, J., Mathis, M., and B. Chandler, "IPv4 Reassembly Errors at High Data Rates", RFC 4963, July 2007.

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

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

[RFC6691] Borman, D., "TCP Options and Maximum Segment Size (MSS)", RFC 6691, July 2012.

[Watson81] Watson, R., "Timer-based Mechanisms in Reliable Transport Protocol Connection Management", Computer Networks, Vol. 5, 1981.

[Zhang86] Zhang, L., "Why TCP Timers Don’t Work Well", Proc. SIGCOMM ’86, Stowe, VT, August 1986.

Appendix A. Implementation Suggestions

TCP Option Layout

The following layouts are recommended for sending options on non-SYN segments, to achieve maximum feasible alignment of 32-bit and 64-bit machines.

```
+--------+--------+--------+--------+
|   NOP  |  NOP   |  TSopt |   10   |
+--------+--------+--------+--------+
|          TSval timestamp          |
+--------+--------+--------+--------+
|          TSecr timestamp          |
+--------+--------+--------+--------+
```

Interaction with the TCP Urgent Pointer

The TCP Urgent pointer, like the TCP window, is a 16 bit value. Some of the original discussion for the TCP Window Scale option included proposals to increase the Urgent pointer to 32 bits. As it turns out, this is unnecessary. There are two observations that should be made:

(1) With IP Version 4, the largest amount of TCP data that can be sent in a single packet is 65495 bytes (64K - 1 -- size of fixed IP and TCP headers).

(2) Updates to the urgent pointer while the user is in "urgent mode" are invisible to the user.

This means that if the Urgent Pointer points beyond the end of the TCP data in the current packet, then the user will remain in urgent mode until the next TCP packet arrives. That packet will update the urgent pointer to a new offset, and the user will never have left urgent mode.

Thus, to properly implement the Urgent Pointer, the sending TCP only has to check for overflow of the 16 bit Urgent Pointer field before filling it in. If it does overflow, than a value of 65535 should be inserted into the Urgent Pointer.

The same technique applies to IP Version 6, except in the case of IPv6 Jumbograms. When IPv6 Jumbograms are supported, [RFC2675] requires additional steps for dealing with the Urgent Pointer, these are described in section 5.2 of [RFC2675].

Appendix B. Duplicates from Earlier Connection Incarnations

There are two cases to be considered: (1) a system crashing (and losing connection state) and restarting, and (2) the same connection being closed and reopened without a loss of host state. These will be described in the following two sections.

B.1. System Crash with Loss of State

TCP's quiet time of one MSL upon system startup handles the loss of connection state in a system crash/restart. For an explanation, see for example "When to Keep Quiet" in the TCP protocol specification [RFC0793]. The MSL that is required here does not depend upon the transfer speed. The current TCP MSL of 2 minutes seemed acceptable as an operational compromise, when many host systems used to take this long to boot after a crash. Current host systems can boot considerably faster.

The timestamp option may be used to ease the MSL requirements (or to provide additional security against data corruption). If timestamps are being used and if the timestamp clock can be guaranteed to be monotonic over a system crash/restart, i.e., if the first value of the sender's timestamp clock after a crash/restart can be guaranteed to be greater than the last value before the restart, then a quiet time is unnecessary.

To dispense totally with the quiet time would require that the host clock be synchronized to a time source that is stable over the crash/restart period, with an accuracy of one timestamp clock tick or better. We can back off from this strict requirement to take advantage of approximate clock synchronization. Suppose that the clock is always re-synchronized to within N timestamp clock ticks and that booting (extended with a quiet time, if necessary) takes more than N ticks. This will guarantee monotonicity of the timestamps, which can then be used to reject old duplicates even without an enforced MSL.

B.2. Closing and Reopening a Connection

When a TCP connection is closed, a delay of 2*MSL in TIME-WAIT state ties up the socket pair for 4 minutes (see Section 3.5 of [RFC0793]. Applications built upon TCP that close one connection and open a new one (e.g., an FTP data transfer connection using Stream mode) must choose a new socket pair each time. The TIME-WAIT delay serves two different purposes:

(a) Implement the full-duplex reliable close handshake of TCP.

The proper time to delay the final close step is not really related to the MSL; it depends instead upon the RTO for the FIN segments and therefore upon the RTT of the path. (It could be argued that the side that is sending a FIN knows what degree of reliability it needs, and therefore it should be able to determine the length of the TIME-WAIT delay for the FIN's recipient. This could be accomplished with an appropriate TCP

option in FIN segments.)

Although there is no formal upper-bound on RTT, common network engineering practice makes an RTT greater than 1 minute very unlikely. Thus, the 4 minute delay in TIME-WAIT state works satisfactorily to provide a reliable full-duplex TCP close. Note again that this is independent of MSL enforcement and network speed.

The TIME-WAIT state could cause an indirect performance problem if an application needed to repeatedly close one connection and open another at a very high frequency, since the number of available TCP ports on a host is less than 2^16. However, high network speeds are not the major contributor to this problem; the RTT is the limiting factor in how quickly connections can be opened and closed. Therefore, this problem will be no worse at high transfer speeds.

(b) Allow old duplicate segments to expire.

To replace this function of TIME-WAIT state, a mechanism would have to operate across connections. PAWS is defined strictly within a single connection; the last timestamp (TS.Recent) is kept in the connection control block, and discarded when a connection is closed.

An additional mechanism could be added to the TCP, a per-host cache of the last timestamp received from any connection. This value could then be used in the PAWS mechanism to reject old duplicate segments from earlier incarnations of the connection, if the timestamp clock can be guaranteed to have ticked at least once since the old connection was open. This would require that the TIME-WAIT delay plus the RTT together must be at least one tick of the sender's timestamp clock. Such an extension is not part of the proposal of this RFC.

Note that this is a variant on the mechanism proposed by Garlick, Rom, and Postel [Garlick77], which required each host to maintain connection records containing the highest sequence numbers on every connection. Using timestamps instead, it is only necessary to keep one quantity per remote host, regardless of the number of simultaneous connections to that host.

Appendix C. Summary of Notation

The following notation has been used in this document.

Options

```
WSopt:            TCP Window Scale Option
TSopt:            TCP Timestamps Option
```

Option Fields

```
shift.cnt:        Window scale byte in WSopt
TSval:            32-bit Timestamp Value field in TSopt
TSecr:            32-bit Timestamp Reply field in TSopt
```

Option Fields in Current Segment

```
SEG.TSval:        TSval field from TSopt in current segment
SEG.TSecr:        TSecr field from TSopt in current segment
SEG.WSopt:        8-bit value in WSopt
```

Clock Values

```
my.TSclock:       System wide source of 32-bit timestamp values
my.TSclock.rate:  Period of my.TSclock (1 ms to 1 sec)
Snd.TSoffset:     A offset for randomizing Snd.TSclock
Snd.TSclock:      my.TSclock + Snd.TSoffset
```

Per-Connection State Variables

```
TS.Recent:        Latest received Timestamp
Last.ACK.sent:    Last ACK field sent
Snd.TS.OK:        1-bit flag
Snd.WS.OK:        1-bit flag
Rcv.Wind.Scale:   Receive window scale power
Snd.Wind.Scale:   Send window scale power
Start.Time:       Snd.TSclock value when segment being timed was
                  sent (used by pre-1323 code).
```

Procedure

```
Update_SRTT(m)    Procedure to update the smoothed RTT and RTT
                  variance estimates, using the rules of
                  [Jacobson88a], given m, a new RTT measurement
```

Appendix D. Pseudo-code Summary

```
Create new TCB => {
    Rcv.wind.scale =
          MIN( 14, MAX(0, floor(log2(receive buffer space)) - 15) );
```

```
        Snd.wind.scale = 0;
        Last.ACK.sent = 0;
        Snd.TS.OK = Snd.WS.OK = FALSE;
        Snd.TSoffset = random 32 bit value
    }

    Send initial <SYN> segment => {
        SEG.WND = MIN( RCV.WND, 65535 );
        Include in segment: TSopt(TSval=Snd.TSclock, TSecr=0);
        Include in segment: WSopt = Rcv.wind.scale;
    }

    Send <SYN,ACK> segment => {
        SEG.ACK = Last.ACK.sent = RCV.NXT;
        SEG.WND = MIN( RCV.WND, 65535 );
        if (Snd.TS.OK) then
              Include in segment:
                    TSopt(TSval=Snd.TSclock, TSecr=TS.Recent);
        if (Snd.WS.OK) then
              Include in segment: WSopt = Rcv.wind.scale;
    }

    Receive <SYN> or <SYN,ACK> segment => {
        if (Segment contains TSopt) then {
              TS.Recent = SEG.TSval;
              Snd.TS.OK = TRUE;
              if (is <SYN,ACK> segment) then
                    Update_SRTT(
                           (Snd.TSclock - SEG.TSecr)/my.TSclock.rate);
        }
        if (Segment contains WSopt) then {
              Snd.wind.scale = SEG.WSopt;
              Snd.WS.OK = TRUE;
              if (the ACK bit is not set, and Rcv.wind.scale has not been
                initialized by the user) then
                    Rcv.wind.scale = Snd.wind.scale;
        }
        else
              Rcv.wind.scale = Snd.wind.scale = 0;
    }

    Send non-SYN segment => {
        SEG.ACK = Last.ACK.sent = RCV.NXT;
        SEG.WND = MIN( RCV.WND >> Rcv.wind.scale, 65535 );
        if (Snd.TS.OK) then
              Include in segment:
                    TSopt(TSval=Snd.TSclock, TSecr=TS.Recent);
    }
```

```
   Receive non-SYN segment in (state >= ESTABLISHED) => {
       Window = (SEG.WND << Snd.wind.scale);
             /* Use 32-bit 'Window' instead of 16-bit 'SEG.WND'
              * in rest of processing.
              */
       if (Segment contains TSopt) then {
             if (SEG.TSval < TS.Recent && Idle less than 24 days) then {
                   if (Send.TS.OK AND (NOT RST) ) then {
                               /* Timestamp too old =>
                                *    segment is unacceptable.
                                */
                         Send ACK segment;
                         Discard segment and return;
                   }
             }
             else {
                   if (SEG.SEQ <= Last.ACK.sent) then
                               TS.Recent = SEG.TSval;
             }
       }
       if (SEG.ACK > SND.UNA) then {
                    /* (At least part of) first segment in
                     * retransmission queue has been ACKed
                     */
             if (Segment contains TSopt) then
                   Update_SRTT(
                          (Snd.TSclock - SEG.TSecr)/my.TSclock.rate);
             else
                   Update_SRTT( /* for compatibility */
                          (Snd.TSclock - Start.Time)/my.TSclock.rate);
       }
   }
```

Appendix E. Event Processing Summary

OPEN Call

...

An initial send sequence number (ISS) is selected. Send a SYN segment of the form:

```
<SEQ=ISS><CTL=SYN><TSval=Snd.TSclock><WSopt=Rcv.Wind.Scale>
```

...

SEND Call

```
      CLOSED STATE (i.e., TCB does not exist)

         ...

      LISTEN STATE

         If the foreign socket is specified, then change the connection
         from passive to active, select an ISS.  Send a SYN segment
         containing the options: <TSval=Snd.TSclock> and
         <WSopt=Rcv.Wind.Scale>.  Set SND.UNA to ISS, SND.NXT to ISS+1.
         Enter SYN-SENT state. ...

      SYN-SENT STATE
      SYN-RECEIVED STATE

         ...

      ESTABLISHED STATE
      CLOSE-WAIT STATE

         Segmentize the buffer and send it with a piggybacked
         acknowledgment (acknowledgment value = RCV.NXT). ...

         If the urgent flag is set ...

         If the Snd.TS.OK flag is set, then include the TCP Timestamps
         option <TSval=Snd.TSclock,TSecr=TS.Recent> in each data
         segment.

         Scale the receive window for transmission in the segment
         header:

                   SEG.WND = (RCV.WND >> Rcv.Wind.Scale).

   SEGMENT ARRIVES

      ...

      If the state is LISTEN then

         first check for an RST

            ...

         second check for an ACK

            ...
```

```
         third check for a SYN

            if the SYN bit is set, check the security.  If the ...

               ...

            if the SEG.PRC is less than the TCB.PRC then continue.

            Check for a Window Scale option (WSopt); if one is found,
            save SEG.WSopt in Snd.Wind.Scale and set Snd.WS.OK flag on.
            Otherwise, set both Snd.Wind.Scale and Rcv.Wind.Scale to
            zero and clear Snd.WS.OK flag.

            Check for a TSopt option; if one is found, save SEG.TSval in
            the variable TS.Recent and turn on the Snd.TS.OK bit.

            Set RCV.NXT to SEG.SEQ+1, IRS is set to SEG.SEQ and any
            other control or text should be queued for processing later.
            ISS should be selected and a SYN segment sent of the form:

                    <SEQ=ISS><ACK=RCV.NXT><CTL=SYN,ACK>

            If the Snd.WS.OK bit is on, include a WSopt option
            <WSopt=Rcv.Wind.Scale> in this segment.  If the Snd.TS.OK
            bit is on, include a TSopt
            <TSval=Snd.TSclock,TSecr=TS.Recent> in this segment.
            Last.ACK.sent is set to RCV.NXT.

            SND.NXT is set to ISS+1 and SND.UNA to ISS.  The connection
            state should be changed to SYN-RECEIVED.  Note that any
            other incoming control or data (combined with SYN) will be
            processed in the SYN-RECEIVED state, but processing of SYN
            and ACK should not be repeated.  If the listen was not fully
            specified (i.e., the foreign socket was not fully
            specified), then the unspecified fields should be filled in
            now.

         fourth other text or control

            ...

      If the state is SYN-SENT then

         first check the ACK bit

            ...
```

```
        ...

        fourth check the SYN bit

           ...

           If the SYN bit is on and the security/compartment and
           precedence are acceptable then, RCV.NXT is set to SEG.SEQ+1,
           IRS is set to SEG.SEQ, and any acknowledgements on the
           retransmission queue which are thereby acknowledged should
           be removed.

           Check for a Window Scale option (WSopt); if it is found,
           save SEG.WSopt in Snd.Wind.Scale; otherwise, set both
           Snd.Wind.Scale and Rcv.Wind.Scale to zero.

           Check for a TSopt option; if one is found, save SEG.TSval in
           variable TS.Recent and turn on the Snd.TS.OK bit in the
           connection control block.  If the ACK bit is set, use
           Snd.TSclock - SEG.TSecr as the initial RTT estimate.

           If SND.UNA > ISS (our SYN has been ACKed), change the
           connection state to ESTABLISHED, form an ACK segment:

                   <SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>

           and send it.  If the Snd.Echo.OK bit is on, include a TSopt
           option <TSval=Snd.TSclock,TSecr=TS.Recent> in this ACK
           segment.  Last.ACK.sent is set to RCV.NXT.

           Data or controls which were queued for transmission may be
           included.  If there are other controls or text in the
           segment then continue processing at the sixth step below
           where the URG bit is checked, otherwise return.

           Otherwise enter SYN-RECEIVED, form a SYN,ACK segment:

                   <SEQ=ISS><ACK=RCV.NXT><CTL=SYN,ACK>

           and send it.  If the Snd.Echo.OK bit is on, include a TSopt
           option <TSval=Snd.TSclock,TSecr=TS.Recent> in this segment.
           If the Snd.WS.OK bit is on, include a WSopt option
           <WSopt=Rcv.Wind.Scale> in this segment.  Last.ACK.sent is
           set to RCV.NXT.

           If there are other controls or text in the segment, queue
           them for processing after the ESTABLISHED state has been
           reached, return.
```

```
         fifth, if neither of the SYN or RST bits is set then drop the
         segment and return.

      Otherwise,

      First, check sequence number

         SYN-RECEIVED STATE
         ESTABLISHED STATE
         FIN-WAIT-1 STATE
         FIN-WAIT-2 STATE
         CLOSE-WAIT STATE
         CLOSING STATE
         LAST-ACK STATE
         TIME-WAIT STATE

            Segments are processed in sequence.  Initial tests on
            arrival are used to discard old duplicates, but further
            processing is done in SEG.SEQ order.  If a segment's
            contents straddle the boundary between old and new, only the
            new parts should be processed.

            Rescale the received window field:

                  TrueWindow = SEG.WND << Snd.Wind.Scale,

            and use "TrueWindow" in place of SEG.WND in the following
            steps.

            Check whether the segment contains a Timestamps option and
            bit Snd.TS.OK is on.  If so:

               If SEG.TSval < TS.Recent and the RST bit is off, then
               test whether connection has been idle less than 24 days;
               if all are true, then the segment is not acceptable;
               follow steps below for an unacceptable segment.

               If SEG.SEQ is less than or equal to Last.ACK.sent, then
               save SEG.TSval in variable TS.Recent.

            There are four cases for the acceptability test for an
            incoming segment:

               ...

            If an incoming segment is not acceptable, an acknowledgment
            should be sent in reply (unless the RST bit is set, if so
            drop the segment and return):
```

```
                    <SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>

            Last.ACK.sent is set to SEG.ACK of the acknowledgment.  If
            the Snd.Echo.OK bit is on, include the Timestamps option
            <TSval=Snd.TSclock,TSecr=TS.Recent> in this ACK segment.
            Set Last.ACK.sent to SEG.ACK and send the ACK segment.
            After sending the acknowledgment, drop the unacceptable
            segment and return.

      ...

      fifth check the ACK field.

         if the ACK bit is off drop the segment and return.

         if the ACK bit is on

            ...

            ESTABLISHED STATE

               If SND.UNA < SEG.ACK <= SND.NXT then, set SND.UNA <-
               SEG.ACK.  Also compute a new estimate of round-trip time.
               If Snd.TS.OK bit is on, use Snd.TSclock - SEG.TSecr;
               otherwise use the elapsed time since the first segment in
               the retransmission queue was sent.  Any segments on the
               retransmission queue which are thereby entirely
               acknowledged...

      ...

      Seventh, process the segment text.

         ESTABLISHED STATE
         FIN-WAIT-1 STATE
         FIN-WAIT-2 STATE

            ...

            Send an acknowledgment of the form:

                    <SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>

            If the Snd.TS.OK bit is on, include Timestamps option
            <TSval=Snd.TSclock,TSecr=TS.Recent> in this ACK segment.
            Set Last.ACK.sent to SEG.ACK of the acknowledgment, and send
            it.  This acknowledgment should be piggy-backed on a segment
            being transmitted if possible without incurring undue delay.
```

...

Appendix F. Timestamps Edge Cases

While the rules laid out for when to calculate RTTM produce the correct results most of the time, there are some edge cases where an incorrect RTTM can be calculated. All of these situations involve the loss of packets. It is felt that these scenarios are rare, and that if they should happen, they will cause a single RTTM measurement to be inflated, which mitigates its effects on RTO calculations.

[Martin03] cites two similar cases when the returning ACK is lost, and before the retransmission timer fires, another returning packet arrives, which ACKs the data. In this case, the RTTM calculated will be inflated:

```
        clock
          tc=1   <A, TSval=1> ------------------->

          tc=2   (lost) <---- <ACK(A), TSecr=1, win=n>
              (RTTM would have been 1)

                 (receive window opens, window update is sent)
          tc=5        <---- <ACK(A), TSecr=1, win=m>
                 (RTTM is calculated at 4)
```

One thing to note about this situation is that it is somewhat bounded by RTO + RTT, limiting how far off the RTTM calculation will be. While more complex scenarios can be constructed that produce larger inflations (e.g., retransmissions are lost), those scenarios involve multiple packet losses, and the connection will have other more serious operational problems than using an inflated RTTM in the RTO calculation.

Appendix G. Changes from RFC 1072, RFC 1185, and RFC 1323

The protocol extensions defined in RFC 1323 document differ in several important ways from those defined in RFC 1072 and RFC 1185.

(a) SACK has been split off into a separate document, [RFC2018].

(b) The detailed rules for sending timestamp replies (see Section 4.4) differ in important ways. The earlier rules could result in an under-estimate of the RTT in certain cases (packets dropped or out of order).

(c) The same value TS.Recent is now shared by the two distinct mechanisms RTTM and PAWS. This simplification became possible because of change (b).

(d) An ambiguity in RFC 1185 was resolved in favor of putting timestamps on ACK as well as data segments. This supports the symmetry of the underlying TCP protocol.

(e) The echo and echo reply options of RFC 1072 were combined into a single Timestamps option, to reflect the symmetry and to simplify processing.

(f) The problem of outdated timestamps on long-idle connections, discussed in Section 5.2.2, was realized and resolved.

(g) RFC 1185 recommended that header prediction take precedence over the timestamp check. Based upon some skepticism about the probabilistic arguments given in Section 5.2.4, it was decided to recommend that the timestamp check be performed first.

(h) The spec was modified so that the extended options will be sent on <SYN,ACK> segments only when they are received in the corresponding <SYN> segments. This provides the most conservative possible conditions for interoperation with implementations without the extensions.

In addition to these substantive changes, the present RFC attempts to specify the algorithms unambiguously by presenting modifications to the Event Processing rules of RFC 793; see Appendix E.

There are additional changes in this document from RFC 1323. These changes are:

(a) The description of which TSecr values can be used to update the measured RTT has been clarified. Specifically, with Timestamps, the Karn algorithm [Karn87] is disabled. The Karn algorithm disables all RTT measurements during retransmission, since it is ambiguous whether the ACK is for the original packet, or the retransmitted packet. With Timestamps, that ambiguity is removed since the TSecr in the ACK will contain the TSval from whichever data packet made it to the destination.

(b) In RFC1323, section 3.4, step (2) of the algorithm to control which timestamp is echoed was incorrect in two regards:

(1) It failed to update TS.recent for a retransmitted segment that resulted from a lost ACK.

(2) It failed if SEG.LEN = 0.

In the new algorithm, the case of SEG.TSval >= TS.recent is included for consistency with the PAWS test.

(c) One correction was made to the Event Processing Summary in Appendix E. In SEND CALL/ESTABLISHED STATE, RCV.WND is used to fill in the SEG.WND value, not SND.WND.

(d) New pseudo-code summary has been added in Appendix D.

(e) Appendix A has been expanded with information about the TCP Urgent Pointer. An earlier revision contained text around the TCP MSS option, which was split off into [RFC6691].

(f) It is now recommended that Timestamps options be included in RST packets if the incoming packet contained a Timestamps option.

(g) RST packets are explicitly excluded from PAWS processing.

(h) Snd.TSoffset and Snd.TSclock variables have been added. Snd.TSclock is the sum of my.TSclock and Snd.TSoffset. This allows the starting points for timestamps to be randomized on a per-connection basis. Setting Snd.TSoffset to zero yields the same results as [RFC1323].

(i) RTTM update processing explicitly excludes packets containing SACK options. This addresses inflation of the RTT during episodes of packet loss in both directions.

(j) In Section 4.2 the if-clause allowing sending of timestamps only when received in a <SYN> or <SYN,ACK> was removed, to allow for late timestamp negotiation.

(k) Section 3.4 was added describing the unavoidable window retraction issue, and explicitly describing the mitigation steps necessary.

(l) Section 2 was added for RFC2119 wording. Normative text was updated with the appropriate phrases.

(m) Removed much of the discussion in Section 1 to streamline the document. However, detailed examples and discussions in Section 3, Section 4 and Section 5 are kept as guideline for implementers.

(n) Moved Appendix "Changes" at the end of the appendices for easier lookup.

Authors' Addresses

David Borman
Quantum Corporation
Mendota Heights MN 55120
USA

Email: david.borman@quantum.com

Bob Braden
University of Southern California
4676 Admiralty Way
Marina del Rey CA 90292
USA

Email: braden@isi.edu

Van Jacobson
Packet Design
2465 Latham Street
Mountain View CA 94040
USA

Email: van@packetdesign.com

Richard Scheffenegger (editor)
NetApp, Inc.
Am Euro Platz 2
Vienna, 1120
Austria

Email: rs@netapp.com

TCP Maintenance and Minor Extensions
(tcpm)
Internet-Draft
Intended status: Informational
Expires: August 15, 2013

M. Kuehlewind, Ed.
University of Stuttgart
R. Scheffenegger
NetApp, Inc.

February 11, 2013

Problem Statement and Requirements for a More Accurate ECN Feedback
draft-ietf-tcpm-accecn-reqs-00

Abstract

Explicit Congestion Notification (ECN) is an IP/TCP mechanism where network nodes can mark IP packets instead of dropping them to indicate congestion to the end-points. An ECN-capable receiver will feedback this information to the sender. ECN is specified for TCP in such a way that only one feedback signal can be transmitted per Round-Trip Time (RTT). Recently, new TCP mechanisms like ConEx or DCTCP need more accurate ECN feedback information in the case where more than one marking is received in one RTT. This documents specifies requirement for different ECN feedback scheme in the TCP header to provide more than one feedback signal per RTT.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction 3
1.1. Requirements Language 3
2. Overview ECN and ECN Nonce in IP/TCP 4
3. Requirements 5
4. Design Approaches 6
4.1. Re-use of Header Bits 6
4.2. Use of Reserved Bits 7
4.3. TCP Option 7
5. Acknowledgements 7
6. IANA Considerations 7
7. Security Considerations 7
8. References 7
8.1. Normative References 7
8.2. Informative References 8
Authors' Addresses 8

1. Introduction

Explicit Congestion Notification (ECN) [RFC3168] is an IP/TCP mechanism where network nodes can mark IP packets instead of dropping them to indicate congestion to the end-points. An ECN-capable receiver will feedback this information to the sender. ECN is specified for TCP in such a way that only one feedback signal can be transmitted per Round-Trip Time (RTT). Recently, proposed mechanisms like Congestion Exposure (ConEx) or DCTCP [Ali10] need more accurate ECN feedback information in case when more than one marking is received in one RTT.

The following scenarios should briefly show where the accurate feedback is needed or provides additional value:

A Standard (RFC5681) TCP sender that supports ConEx:
In this case the congestion control algorithm still ignores multiple marks per RTT, while the ConEx mechanism uses the extra information per RTT to re-echo more precise congestion information.

A sender using DCTCP congestion control without ConEx:
The congestion control algorithm uses the extra info per RTT to perform its decrease depending on the number of congestion marks.

A sender using DCTCP congestion control and supports ConEx:
Both the congestion control algorithm and ConEx use the accurate ECN feedback mechanism.

A standard TCP sender (using RFC5681 congestion control algorithm) without ConEx:
No accurate feedback is necessary here. The congestion control algorithm still react only on one signal per RTT.
But it is best to have one generic feedback mechanism, whether it is used or not.

This documents ...

1.1. Requirements Language

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 RFC 2119 [RFC2119].

We use the following terminology from [RFC3168] and [RFC3540]:

The ECN field in the IP header:

CE: the Congestion Experienced codepoint, and

ECT(0): the first ECN-Capable Transport codepoint, and

ECT(1): the second ECN-Capable Transport codepoint.

The ECN flags in the TCP header:

CWR: the Congestion Window Reduced flag,

ECE: the ECN-Echo flag, and

NS: ECN Nonce Sum.

In this document, we will call the ECN feedback scheme as specified in [RFC3168] the 'classic ECN' and our new proposal the 'more accurate ECN feedback' scheme. A 'congestion mark' is defined as an IP packet where the CE codepoint is set. A 'congestion event' refers to one or more congestion marks belong to the same overload situation in the network (usually during one RTT).

2. Overview ECN and ECN Nonce in IP/TCP

ECN requires two bits in the IP header. The ECN capability of a packet is indicated when either one of the two bits is set. An ECN sender can set one or the other bit to indicate an ECN-capable transport (ECT) which results in two signals, ECT(0) and ECT(1). A network node can set both bits simultaneously when it experiences congestion. When both bits are set the packet is regarded as "Congestion Experienced" (CE).

In the TCP header the first two bits in byte 14 are defined for the use of ECN. The TCP mechanism for signaling the reception of a congestion mark uses the ECN-Echo (ECE) flag in the TCP header. To enable the TCP receiver to determine when to stop setting the ECN-Echo flag, the CWR flag is set by the sender upon reception of the feedback signal. This leads always to a full RTT of ACKs with ECE set. Thus any additional CE markings arriving within this RTT can not signaled back anymore.

ECN-Nonce [RFC3540] is an optional addition to ECN that is used to protect the TCP sender against accidental or malicious concealment of marked or dropped packets. This addition defines the last bit of

byte 13 in the TCP header as the Nonce Sum (NS) bit. With ECN-Nonce a nonce sum is maintain that counts the occurrence of ECT(1) packets.

```
 0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|               |           | N | C | E | U | A | P | R | S | F |
| Header Length | Reserved  | S | W | C | R | C | S | S | Y | I |
|               |           |   | R | E | G | K | H | T | N | N |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Figure 1: The (post-ECN Nonce) definition of the TCP header flags

3. Requirements

The requirements of the accurate ECN feedback protocol for the use of e.g. Conex or DCTCP are to have a fairly accurate (not necessarily perfect), timely and protected signaling. This leads to the following requirements:

Resilience
The ECN feedback signal is carried within the TCP acknowledgment. TCP ACKs can get lost. Moreover, delayed ACK are mostly used with TCP. That means in most cases only every second data packets triggers an ACK. In a high congestion situation where most of the packet are marked with CE, an accurate feedback mechanism must still be able to signal sufficient congestion information. Thus the accurate ECN feedback extension has to take delayed ACK and ACK loss into account.

Timely
The CE marking is induced by a network node on the transmission path and echoed by the receiver in the TCP acknowledgment. Thus when this information arrives at the sender, its naturally already about one RTT old. With a sufficient ACK rate a further delay of a small number of ACK can be tolerated but with large delays this information will be out dated due to high dynamic in the network. TCP congestion control which introduces parts of these dynamics operates on a time scale of one RTT. Thus the congestion feedback information should be delivered timely (within one RTT).

Integrity
With ECN Nonce, a misbehaving receiver or network node can be detected with a certain probability. As this accurate ECN feedback is reusing the NS bit, it is encouraged to ensure

integrity as least as good as ECN Nonce. If this is not possible, alternative approaches should be provided how a mechanism using the accurate ECN feedback extension can re-ensure integrity or give strong incentives for the receiver and network node to cooperate honestly.

Accuracy
Classic ECN feeds back one congestion notification per RTT, as this is supposed to be used for TCP congestion control which reduces the sending rate at most once per RTT. The accurate ECN feedback scheme has to ensure that if a congestion events occurs at least one congestion notification is echoed and received per RTT as classic ECN would do. Of course, the goal of this extension is to reconstruct the number of CE marking more accurately. However, a sender should not assume to get the exact number of congestion marking in all situations.

Complexity
Of course, the more accurate ECN feedback can also be used, even if only one ECN feedback signal per RTT is need. The implementation should be as simple as possible and only a minimum of addition state information should be needed. A proposal fulfilling this for a more accurate ECN feedback can then also be the standard ECN feedback mechanism.

4. Design Approaches

4.1. Re-use of Header Bits

The idea is to use the ECE, CWR and NS bits for additional capability negotiation during the TCP handshake exchange, and then for the more accurate ECN feedback itself on subsequent packets in the flow (where SYN is not set). This appraoch only provide a limited resiliency against ACK lost.

There have been several codings proposed so far: The one bit scheme sends one ECE for each CE received (+ redundancy in next ACK using the CWR bit). The 3 bit counter scheme uses all three bits for continuesly feeding the three most significant bits of a CE counter back. The 3 bit codepoint scheme encodes either a CE counter or an ECT(1) counter in 8 codepoints.

Discussion on ACK loss and ECN...

ToDo: Use of other header bit?

4.2. Use of Reserved Bits

As seen in Figure 1, there are currently three unused flag bits in the TCP header. The proposed scheme could be extended by one or more bits, to add higher resiliency against ACK loss. The relative gain would be proportionally higher resiliency against ACK loss, while the respective drawbacks would remain identical.

4.3. TCP Option

Alternatively, a new TCP option could be introduced, to help maintain the accuracy, and integrity of the ECN feedback between receiver and sender. Such an option could provide more information. E.g. ECN for RTP/UDP provides explicit the number of ECT(0), ECT(1), CE, non-ECT marked and lost packets. However, deploying new TCP options has its own challenges. A separate document proposes a new TCP Option for accurate ECN feedback [I-D.kuehlewind-tcpm-accurate-ecn-option]. This option could be used in addition to a more accurate ECN feedback scheme described here or in addition to classic ECN, when available and needed.

5. Acknowledgements

6. IANA Considerations

This memo includes no request to IANA.

7. Security Considerations

TBD

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, September 2001.

[RFC3540] Spring, N., Wetherall, D., and D. Ely, "Robust Explicit Congestion Notification (ECN) Signaling with Nonces",

RFC 3540, June 2003.

8.2. Informative References

[Ali10] Alizadeh, M., Greenberg, A., Maltz, D., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and M. Sridharan, "DCTCP: Efficient Packet Transport for the Commoditized Data Center", Jan 2010.

[I-D.briscoe-tsvwg-re-ecn-tcp]
Briscoe, B., Jacquet, A., Moncaster, T., and A. Smith, "Re-ECN: Adding Accountability for Causing Congestion to TCP/IP", draft-briscoe-tsvwg-re-ecn-tcp-09 (work in progress), October 2010.

[I-D.kuehlewind-tcpm-accurate-ecn-option]
Kuehlewind, M. and R. Scheffenegger, "Accurate ECN Feedback Option in TCP", draft-kuehlewind-tcpm-accurate-ecn-option-01 (work in progress), July 2012.

[RFC5562] Kuzmanovic, A., Mondal, A., Floyd, S., and K. Ramakrishnan, "Adding Explicit Congestion Notification (ECN) Capability to TCP's SYN/ACK Packets", RFC 5562, June 2009.

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

[RFC5690] Floyd, S., Arcia, A., Ros, D., and J. Iyengar, "Adding Acknowledgement Congestion Control to TCP", RFC 5690, February 2010.

Authors' Addresses

Mirja Kuehlewind (editor)
University of Stuttgart
Pfaffenwaldring 47
Stuttgart 70569
Germany

Email: mirja.kuehlewind@ikr.uni-stuttgart.de

Richard Scheffenegger
NetApp, Inc.
Am Euro Platz 2
Vienna, 1120
Austria

Phone: +43 1 3676811 3146
Email: rs@netapp.com

TCPM Working Group
Internet Draft
Intended status: Proposed Standard
Expires: December 2013

J. Touch
USC/ISI

June 4, 2013

Shared Use of Experimental TCP Options
draft-ietf-tcpm-experimental-options-06.txt

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), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html

This Internet-Draft will expire on December 4, 2013.

Copyright Notice

Abstract

This document describes how the experimental TCP option codepoints can concurrently support multiple TCP extensions, even within the same connection. It uses a new IANA TCP experiment identifier, and is also robust to experiments that are not registered and those that do not use this sharing mechanism. It is recommended for all new TCP options that use these codepoints.

Table of Contents

1. Introduction.....2
2. Conventions used in this document.....4
3. TCP Experimental Option Structure.....4
   3.1. Selecting an ExID.....6
   3.2. Impact on TCP Option Processing.....7
4. Reducing the Impact of False Positives.....7
5. Migration to Assigned Options.....8
6. Rationale.....8
7. Security Considerations.....9
8. IANA Considerations.....9
9. References.....10
   9.1. Normative References.....10
   9.2. Informative References.....10
10. Acknowledgments.....12

1. Introduction

TCP includes options to enable new protocol capabilities that can be activated only where needed and supported [RFC793]. The space for identifying such options is small - 256 values, of which 30 are assigned at the time this document was published [IANA]. Two of these codepoints are allocated to support experiments (253, 254) [RFC4727]. These values are intended for testing purposes or anytime an assigned codepoint is either not warranted or available, e.g., based on the maturity status of the defined capability (i.e., Experimental or Informational, rather than Standards Track).

The term "experimental TCP options" refers here to options that use the TCP experimental option codepoints [RFC4727]. Such experiments can be described in any type of RFC - Experimental, Informational, etc., and are intended to be used both in controlled environments

and in are allowed in public deployments (when not enabled as default) [RFC3692]. Nothing prohibits deploying multiple experiments in the same environment - controlled or public. Further, some protocols are specified in Experimental or Informational RFCs, which either include parameters or design choices not yet understood or which might not be widely deployed [RFC2026]. TCP options in such RFCs are typically not eligible for assigned TCP option codepoints [RFC2780], and so there is a need to share use of the experimental option codepoints.

There is currently no mechanism to support shared use of the TCP experimental option codepoints, either by different experiments on different connections, or for more than two experimental options in the same connection. Experimental options 253 and 254 are already deployed in operational code to support an early version of TCP authentication. Option 253 is also documented for the experimental TCP Cookie Transaction option [RFC6013]. This shared use results in collisions in which a single codepoint can appear multiple times in a single TCP segment and for which each use is ambiguous.

Other codepoints have been used without assignment (known as "squatting"), notably 31-32 (TCP cookie transactions, as originally distributed and in its API doc) and 76-78 (tcpcrypt) [Bi11][Si11]. Commercial products reportedly also use unassigned options 33, 69-70, and 76-78 as well. Even though these uses are unauthorized, they currently impact legitimate assignees.

Both such misuses (squatting on both experimental and assigned codepoints) are expected to continue, but there are several approaches which can alleviate the impact on cooperating protocol designers. One proposal relaxes the requirements for assignment of TCP options, allowing them to be assigned more readily for protocols that have not been standardized through the IETF process [RFC5226]. Another proposal assigns a larger pool to the TCP experiment option codepoints and manages their sharing through IANA coordination [Ed11].

The approach proposed in this document does not require additional TCP option codepoints, and is robust to those who choose either not to support it or not to register their experiments. The solution adds a field to the structure of the experimental TCP option. This field is populated with an "experiment identifier" (ExID) defined as part of a specific option experiment. The ExID helps reduce the probability of a collision of independent experimental uses of the same option codepoint, both for those who follow this document (using registered ExIDs) and those who do not (squatters who either ignore this extension or do not register their ExIDs).

The solution proposed in this document is recommended for all new protocols that use TCP experimental option codepoints. The techniques used here may also help share other experimental codepoints, but that issue is out of scope for this document.

2. Conventions used in this document

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 RFC-2119 [RFC2119].

In this document, these words will appear with that interpretation only when in ALL CAPS. Lower case uses of these words are not to be interpreted as carrying RFC-2119 significance.

In this document, the characters ">>" preceding an indented line(s) indicates a compliance requirement statement using the key words listed above. This convention aids reviewers in quickly identifying or finding the explicit compliance requirements of this RFC.

3. TCP Experimental Option Structure

TCP options have the current common structure [RFC793], in which the first byte is the codepoint (Kind) and the second byte is the length of the option in bytes (Length):

```
 0          1          2          3
 01234567 89012345 67890123 45678901
+--------+--------+--------+--------+
|  Kind  | Length |       ...       |
+--------+--------+--------+--------+
|    ...
+--------
```

Figure 1 TCP Option Structure [RFC793]

This document extends the option structure for experimental codepoints (253, 254) with an experiment identifier (ExID), which is either 2 or 4 bytes in length. The ExID is used to differentiate different experiments, and is the first field after the Kind and Length, as follows:

```
 0          1          2          3
 01234567 89012345 67890123 45678901
+--------+--------+--------+--------+
|  Kind  | Length |       ExID      |
+--------+--------+--------+--------+
|  option contents...
+--------+--------+--------+---
```

Figure 2 TCP Experimental Option with a 16-bit ExID

```
 0          1          2          3
 01234567 89012345 67890123 45678901
+--------+--------+--------+--------+
|  Kind  | Length |       ExID      |
+--------+--------+--------+--------+
|   ExID (con't)  |  option contents...
+--------+--------+--------+---
```

Figure 3 TCP Experimental Option with a 32-bit ExID

This mechanism is encouraged for all TCP options that are not yet eligible for assigned codepoints:

>> Protocols requiring new TCP option codepoints that are not eligible for assigned values SHOULD use the existing TCP experimental option codepoints (253, 254) with ExIDs as described in this document.

This mechanism is encouraged for all TCP options using the current experimental codepoints in controlled environments:

>> All protocols using the TCP experimental option codepoints (253, 254), even those deployed in controlled environments, SHOULD use ExIDs as described in this document.

This mechanism is required for all TCP options using the current experimental codepoints that are publicly deployed, whether enabled by default or not:

>> All protocols using the TCP experimental option codepoints (253, 254) that are deployed outside controlled environments, such as in the public Internet, MUST use ExIDs as described in this document.

Once a TCP option uses the mechanism in this document, registration of the ExID with IANA is required:

>> All protocols using ExIDs as described in this document MUST register those ExIDs with IANA.

Applicants register their desired ExID by contacting IANA [IANA].

3.1. Selecting an ExID

ExIDs are selected at design time, when the protocol designer first implements or specifies the experimental option. ExIDs can be either 16-bits or 32-bits. In both cases, the value is stored in the header in network-standard (big-endian) byte order. ExIDs combine properties of IANA registered codepoints with "magic numbers".

>> All ExIDs MUST be either 16-bits or 32-bits long.

Use of the ExID, whether 16-bit or 32-bit, helps reduce the probability of a false positive collision with those who either do not register their experiment or who do not implement this mechanism.

In order to conserve TCP option space, either for use within a specific option or to be available for other options:

>> Options implementing the mechanism of this document SHOULD use 16-bit ExIDs except where explicitly motivating the need for 32-bit ExIDs, e.g., to avoid false positives or maintain alignment with an expected future assigned codepoint.

ExIDs are registered with IANA using "first-come, first-served" priority based on the first two bytes. Those two bytes are thus sufficient to interpret which experimental option is contained in the option field.

>> All ExIDs MUST be unique based on their first 16 bits.

The second two bytes serve as a "magic number". A magic number is a self-selected codepoint whose primary value is its unlikely collision with values selected by others. Magic numbers are used in other protocols, e.g., BOOTP [RFC951] and DHCP [RFC2131].

Using the additional magic number bytes helps the option contents have the same byte alignment in the TCP header as they would have if (or when) a conventional (non-experiment) TCP option codepoint is assigned. Use of the same alignment reduces the potential for implementation errors, especially in using the same word-alignment padding, if the same software is later modified to use a conventional codepoint. Use of the longer, 32-bit ExID further

decreases the probability of such a false positive compared to those using shorter, 16-bit ExIDs.

Use of the ExID does consume TCP option space but enables concurrent use of the experimental codepoints and provides protection against false positives, leaving less space for other options (including other experiments). Use of the longer, 32-bit ExID consumes more space, but provides more protection against false positives.S

3.2. Impact on TCP Option Processing

The ExID number is considered part of the TCP option, not the TCP option header. The presence of the ExID increases the effective option Length field by the size of the ExID. The presence of this ExID is thus transparent to implementations that do not support TCP options where it is used.

During TCP processing, ExIDs in experimental options are matched against the ExIDs for each implemented protocol. The remainder of the option is specified by the particular experimental protocol.

>> Experimental options that have ExIDs that do not match implemented protocols MUST be ignored.

The ExID mechanism must be coordinated during connection establishment, just as with any TCP option.

>> TCP ExID, if used in any TCP segment of a connection, MUST be present in TCP SYN segments of that connection.

>> TCP experimental option ExIDS, if used in any TCP segment of a connection, SHOULD be used in all TCP segments of that connection in which any experimental option is present.

Use of an ExID uses additional space in the TCP header and requires additional protocol processing by experimental protocols. Because these are experiments, neither consideration is a substantial impediment; a finalized protocol can avoid both issues with the assignment of a dedicated option codepoint later.

4. Reducing the Impact of False Positives

False positives occur where the registered ExID of an experiment matches the value of an option that does not use ExIDs. Such collisions can cause an option to be interpreted by the incorrect processing routine. Use of checksums or signatures may help an

experiment use the shorter ExID while reducing the corresponding increased potential for false positives.

>> Experiments that are not robust to ExID false positives SHOULD implement other detection measures, such as checksums or minimal digital signatures over the experimental options they support.

5. Migration to Assigned Options

Some experiments may transition from experiment, and become eligible for an assigned TCP option codepoint. This document does not recommend a specific migration plan to transition from use of the experimental TCP options/ExIDs to use of an assigned codepoint.

However, once an assigned codepoint is allocated, use of an ExID represents unnecessary overhead. As a result:

>> Once a TCP option codepoint is assigned to a protocol, that protocol SHOULD NOT continue to use an ExID as part of that assigned codepoint.

This document does not recommend whether or how an implementation of an assigned codepoint can be backward-compatible with use of the experimental codepoint/ExID.

However, some implementers may be tempted to include both the experimental and assigned codepoint in the same segment, e.g., in a SYN to support backward-compatibility during connection establishment. This is a poor use limited resources and so to ensure conservation of the TCP option space:

>> A TCP segment MUST NOT contain both an assigned TCP option codepoint and a TCP experimental option codepoint for the same protocol.

Instead, a TCP that intends backward compatibility might send multiple SYNs with alternates of the same option and discard all but the most desired successful connection. Although this approach may resolve more slowly or require additional effort at the endpoints, it is preferable to excessively consuming TCP option space.

6. Rationale

The ExIDs described in this document combine properties of IANA first-come/first-served (FCFS) registered values with magic numbers. Although IANA FCFS registries are common, so too are those who either fail to register or who 'squat' by deliberately using

codepoints that are assigned to others. The approach in this document is intended to recognize this reality and be more robust to its consequences than would be a conventional IANA FCFS registry.

Existing ID spaces were considered as ExIDs in the development of this mechanism, including IEEE Organizationally Unique Identifier (OUI) and IANA Private Enterprise Numbers (PENs) [802] [OUI] [RFC1155].

OUIs are 24-bit identifiers that are combined with 24 to 40-bits of privately-assigned space to create identifiers that commonly assigned to a unique piece of hardware. OUIs are already longer than the smaller ExID value, and obtaining an OUI is costly (currently $1,885.00 USD). An OUI could be obtained for each experiment, but this could be considered expensive. An OUI already assigned to an organization could be shared if extended (to support multiple experiments within an organization), but this would either require coordination within an organization or an IANA registry; the former is prohibitive, and the latter is more complicated than to have IANA manage the entire space.

PENs were originally used in SNMP [RFC1157]. PENs are identifiers that can be obtained without cost from IANA [PEN]. Despite the current registry, the size of the PEN assignment space is currently undefined, and has only recently been proposed (as 32-bits) [Li12]. PENs are currently assigned to organizations, and there is no current process for assigning them to individuals. Finally, if 32-bits as expected, they would be larger than needed in many cases.

7. Security Considerations

The mechanism described in this document is not intended to provide (nor does it weaken existing) security for TCP option processing.

8. IANA Considerations

This document calls for IANA to create a new TCP experimental option Experiment Identifier (ExID) registry. The registry records both 16-bit and 32-bit ExIDs, as well as a name and e-mail contact for each entry. ExIDs are registered for use with both TCP experimental option codepoints, i.e., with TCP options with values of 253 and 254.

Entries are assigned on a First-Come, First-Served (FCFS) basis [RFC5226]. The registry operates FCFS on the first two bytes of the ExID (in network-standard order) but records the entire ExID (in network-standard order). Some examples are:

- o 0x12340000 collides with a previous registration of 0x1234abcd
- o 0x5678 collides with a previous registration of 0x56780123
- o 0xabcd1234 collides it a previous registration of 0xabcd

IANA will advise applicants of duplicate entries to select an alternate value, as per typical FCFS processing.

IANA will record known duplicate uses to assist the community in both debugging assigned uses as well as correcting unauthorized duplicate uses.

IANA should impose no requirements on making a registration other than indicating the desired codepoint and providing a point of contact. A short description or acronym for the use is desired, but should not be required.

9. References

9.1. Normative References

[RFC793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, Sep. 1981.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC4727] Fenner, B., "Experimental Values in IPv4, IPv6, ICMPv4, ICMPv6, UDP, and TCP Headers", RFC 4727, Nov. 2006.

[RFC5226] Narten, T., H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

9.2. Informative References

[802] "IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture", IEEE 802-2001, 8 March 2002.

[Bi11] Bittau, A., D. Boneh, M. Hamburg, M. Handley, D. Mazieres, Q. Slack, "Cryptographic protection of TCP Streams (tcpcrypt)", work in progress, draft-bittau-tcp-crypt-03, Sep. 3, 2012.

[Ed11] Eddy, W., "Additional TCP Experimental-Use Options", work in progress, draft-eddy-tcpm-addl-exp-options-00, Aug. 16, 2011.

[IANA] IANA web pages, http://www.iana.org/

[Li12] Liang, P., A. Melnikov, "Private Enterprise Number (PEN) practices and Internet Assigned Numbers: Authority (IANA) considerations for registration procedures", draft-liang-iana-pen-01, (work in progress), June 2012.

[OUI] IEEE OUI registry, http://standards.ieee.org/develop/regauth/oui/

[PEN] IANA Private Enterprise Numbers, http://www.iana.org/assignments/enterprise-numbers

[RFC951] Croft, B., J. Gilmore, "BOOTSTRAP PROTOCOL (BOOTP)", RFC 951, Sept. 1985.

[RFC1155] Rose, M., K. McCloghrie, "Structure and Identification of Management Information for TCP/IP-based internets", RFC 1155, May 1990.

[RFC1157] Case, J., M. Fedor, M. Schoffstall, J. Davin, "A Simple Network Management Protocol (SNMP)", RFC 1157, May 1990.

[RFC2026] Bradner, S., "The Internet Standards Process -- Revision 3", BCP 9, RFC 2026, Oct. 1996.

[RFC2131] Droms, R., "Dynamic Host Configuration Protocol", RFC 2131, Mar. 1997.

[RFC2780] Bradner, S., V. Paxson, "IANA Allocation Guidelines For Values In the Internet Protocol and Related Headers", BCP 37, RFC 2780, Mar. 2000.

[RFC3692] Narten, T., "Assigning Experimental and Testing Numbers Considered Useful", BCP 82, RFC 3692, Jan. 2004.

[RFC6013] Simpson, W., "TCP Cookie Transactions (TCPCT)", RFC 6013, Jan. 2011.

[Si11] Simpson, W., "TCP Cookie Transactions (TCPCT) Sockets Application Program Interface (API)", work in progress, draft-simpson-tcpct-api-04, Apr. 7, 2011.

10. Acknowledgments

This document was motivated by discussions on the IETF TCPM mailing list and by Wes Eddy's proposal [Ed11]. Yoshifumi Nishida, Pasi Sarolathi, and Michael Scharf provided detailed feedback.

This document was prepared using 2-Word-v2.0.template.dot.

Authors' Addresses

Joe Touch
USC/ISI
4676 Admiralty Way
Marina del Rey, CA 90292-6695 U.S.A.

Phone: +1 (310) 448-9151
Email: touch@isi.edu

Internet Draft
draft-ietf-tcpm-fastopen-02.txt
Intended status: Experimental
Expiration date: April, 2013

Y. Cheng
J. Chu
S. Radhakrishnan
A. Jain
Google, Inc.
Octobor 22, 2012

TCP Fast Open

Status of this Memo

Distribution of this memo is unlimited.

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), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/1id-abstracts.html

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html

This Internet-Draft will expire in August, 2012.

Copyright Notice

Abstract

TCP Fast Open (TFO) allows data to be carried in the SYN and SYN-ACK packets and consumed by the receiving end during the initial connection handshake, thus saving up to one full round trip time (RTT) compared to standard TCP which requires a three-way handshake (3WHS) to complete before data can be exchanged.

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 RFC 2119 [RFC2119]. TFO refers to TCP Fast Open. Client refers to the TCP's active open side and server refers to the TCP's passive open side.

1. Introduction

TCP Fast Open (TFO) enables data to be exchanged safely during TCP's connection handshake.

This document describes a design that enables applications to save a round trip while avoiding severe security ramifications. At the core of TFO is a security cookie used by the server side to authenticate a client initiating a TFO connection. This document covers the details of exchanging data during TCP's initial handshake, the protocol for TFO cookies, and potential new security vulnerabilities and their mitigation. It also includes discussions of deployment issues and related proposals. TFO requires extensions to the socket API but this document does not cover that.

TFO is motivated by the performance needs of today's Web applications. Network latency is determined by the round-trip time (RTT) and the number of round trips required to transfer application data. RTT consists of propagation delay and queuing delay. Network bandwidth has grown substantially over the past two decades, reducing queuing delay, while propagation delay is largely constrained by the speed of light and has remained unchanged. Therefore reducing the number of round trips has become the most effective way to improve the latency of Web applications [CDCM11].

Standard TCP only permits data exchange after 3WHS [RFC793], which adds one RTT to the network latency. For short transfers (e.g., web objects) this additional RTT is a significant portion of the network latency [THK98]. One widely deployed solution is HTTP persistent connections. However, this solution is limited since hosts and middle boxes terminate idle TCP connections due to resource constraints. For example, the Chrome browser keeps TCP connections idle up to 5

minutes but 35% of Chrome HTTP requests are made on new TCP connections. We discuss HTTP persistent connections further in section 7.1.

2. Data In SYN

[RFC793] (section 3.4) already allows data in SYN packets but forbids the receiver to deliver the data to the application until 3WHS is completed. This is because TCP's initial handshake serves to capture 1) Old or duplicate SYNs and 2)SYNs with spoofed IP addresses.

TFO allows data to be delivered to the application before 3WHS is completed, thus opening itself to a possible data integrity problem caused by the problematic SYN packets above. This could cause a problem in the following two examples: a) the receiver host receives both duplicate and original SYNs before and after the host reboots, and b) the duplicate is received after the connection created by the original SYN has been closed. The receiver will not be protected by the 2MSL TIMEWAIT state if the close is initiated by the sender. In both cases, the data is replayed.

2.1. TCP Semantics and Duplicate SYNs

The proposed T/TCP protocol employs a new TCP "TAO" option and connection count to guard against old or duplicate SYNs [RFC1644]. The solution is complex, involving state tracking on a per remote peer basis, and is vulnerable to IP spoofing attacks. Moreover, it has been shown that despite its complexity, T/TCP is still not entirely protected. Old or duplicate SYNs may still be accepted by a T/TCP server [PHRACK98].

Rather than trying to capture all dubious SYN packets to make TFO 100% compatible with TCP semantics, we made a design decision early on to accept old SYN packets with data, i.e., to restrict TFO to use with a class of applications that are tolerant of duplicate SYN packets with data. We believe this is the right design trade-off balancing complexity with usefulness. Applications that require transactional semantics already deploy specific mechanisms to tolerate similar data replay issues in TCP today. For example, a browser reload event may replay any HTTP request even without data in SYN. For transactional HTTP requests applications typically include unique identifiers in the HTTP headers. Thus, allowing data in SYN poses little risk to existing HTTP applications.

However, we note that some applications may rely on TCP 3-way handshake semantics. For this reason, TFO MUST be used explicitly by applications on a per service port basis.

2.2. SYNs with spoofed IP addresses

Standard TCP suffers from the SYN flood attack [RFC4987] because bogus SYN packets, i.e., SYN packets with spoofed source IP addresses can easily fill up a listener's small queue, causing a service port to be blocked completely until timeouts. Secondary damage comes from these SYN requests taking up memory space. Though this is less of an issue today as servers typically have plenty of memory.

TFO goes one step further to allow server side TCP to process and send up data to the application layer before 3WHS is completed. This opens up more serious new vulnerabilities. Applications serving ports that have TFO enabled may waste lots of CPU and memory resources processing the requests and producing the responses. If the response is much larger than the request, the attacker can mount an amplified reflection attack against victims of choice beyond the TFO server itself.

Numerous mitigation techniques against the regular SYN flood attack exist and have been well documented [RFC4987]. Unfortunately none are applicable to TFO. We propose a server supplied cookie to mitigate most of the security issues introduced by TFO. We defer further discussion of SYN flood attacks to the "Security Considerations" section.

3. Protocol Overview

The key component of TFO is the Fast Open Cookie (cookie), a message authentication code (MAC) tag generated by the server. The client requests a cookie in one regular TCP connection, then uses it for future TCP connections to exchange data during 3WHS:
Requesting a Fast Open Cookie:

1. The client sends a SYN with a Fast Open Cookie Request option.

2. The server generates a cookie and sends it through the Fast Open Cookie option of a SYN-ACK packet.

3. The client caches the cookie for future TCP Fast Open connections (see below).

Performing TCP Fast Open:

1. The client sends a SYN with Fast Open Cookie option and data.

2. The server validates the cookie:
   a. If the cookie is valid, the server sends a SYN-ACK acknowledging both the SYN and the data. The server then delivers the data to the application.

b. Otherwise, the server drops the data and sends a SYN-ACK acknowledging only the SYN sequence number.

3. If the server accepts the data in the SYN packet, it may send the response data before the handshake finishes. The max amount is governed by the TCP's congestion control [RFC5681].

4. The client sends an ACK acknowledging the SYN and the server data. If the client's data is not acknowledged, the client retransmits the data in the ACK packet.

5. The rest of the connection proceeds like a normal TCP connection.

The client can repeat many Fast Open operations once it acquires a cookie (until the cookie is expired by the server). Thus TFO is useful for applications that have temporal locality on client and server connections.

Requesting Fast Open Cookie in connection 1:

```
   TCP A (Client)                                      TCP B(Server)
   ______________                                      _____________
   CLOSED                                                     LISTEN

#1 SYN-SENT       ----- <SYN,CookieOpt=NIL>  ---------->  SYN-RCVD

#2 ESTABLISHED    <---- <SYN,ACK,CookieOpt=C> ----------  SYN-RCVD
    (caches cookie C)
```

Performing TCP Fast Open in connection 2:

```
   TCP A (Client)                                      TCP B(Server)
   ______________                                      _____________
   CLOSED                                                     LISTEN

#1 SYN-SENT       ----- <SYN=x,CookieOpt=C,DATA_A> ---->  SYN-RCVD

#2 ESTABLISHED    <---- <SYN=y,ACK=x+len(DATA_A)+1> ----  SYN-RCVD

#3 ESTABLISHED    <---- <ACK=x+len(DATA_A)+1,DATA_B>----  SYN-RCVD

#4 ESTABLISHED    ----- <ACK=y+1>--------------------> ESTABLISHED

#5 ESTABLISHED    --- <ACK=y+len(DATA_B)+1>----------> ESTABLISHED
```

4. Protocol Details

4.1. Fast Open Cookie

The Fast Open Cookie is designed to mitigate new security vulnerabilities in order to enable data exchange during handshake. The cookie is a message authentication code tag generated by the server and is opaque to the client; the client simply caches the cookie and passes it back on subsequent SYN packets to open new connections. The server can expire the cookie at any time to enhance security.

4.1.1. TCP Options

Fast Open Cookie Option

The server uses this option to grant a cookie to the client in the SYN-ACK packet; the client uses it to pass the cookie back to the server in the SYN packet.

```
                                +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                                |      Kind     |    Length     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                            Cookie                             ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
Kind            1 byte: constant TBD (assigned by IANA)
Length          1 byte: range 6 to 18 (bytes); limited by
                        remaining space in the options field.
                        The number MUST be even.
Cookie          4 to 16 bytes (Length - 2)
```

Options with invalid Length values or without SYN flag set MUST be ignored. The minimum Cookie size is 4 bytes. Although the diagram shows a cookie aligned on 32-bit boundaries, alignment is not required.

Fast Open Cookie Request Option

The client uses this option in the SYN packet to request a cookie from a TFO-enabled server

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Kind     |    Length     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
Kind            1 byte: same as the Fast Open Cookie option
Length          1 byte: constant 2. This distinguishes the option
                        from the Fast Open cookie option.
```

Options with invalid Length values, without SYN flag set, or with ACK flag set MUST be ignored.

4.1.2. Server Cookie Handling

The server is in charge of cookie generation and authentication. The cookie SHOULD be a message authentication code tag with the following properties:

1. The cookie authenticates the client's (source) IP address of the SYN packet. The IP address can be an IPv4 or IPv6 address.

2. The cookie can only be generated by the server and can not be fabricated by any other parties including the client.

3. The generation and verification are fast relative to the rest of SYN and SYN-ACK processing.

4. A server may encode other information in the cookie, and accept more than one valid cookie per client at any given time. But this is all server implementation dependent and transparent to the client.

5. The cookie expires after a certain amount of time. The reason for cookie expiration is detailed in the "Security Consideration" section. This can be done by either periodically changing the server key used to generate cookies or including a timestamp when generating the cookie.

   To gradually invalidate cookies over time, the server can implement key rotation to generate and verify cookies using multiple keys. This approach is useful for large-scale servers to retain Fast Open rolling key updates. We do not specify a particular mechanism because the implementation is often server specific.

The server supports the cookie generation and verification operations:

- GetCookie(IP_Address): returns a (new) cookie

- IsCookieValid(IP_Address, Cookie): checks if the cookie is valid, i.e., it has not expired and it authenticates the client IP address.

Example Implementation: a simple implementation is to use AES_128 to encrypt the IPv4 (with padding) or IPv6 address and truncate to 64 bits. The server can periodically update the key to expire the cookies. AES encryption on recent processors is fast and takes only a few hundred nanoseconds [RCCJB11].

If only one valid cookie is allowed per-client and the server can regenerate the cookie independently, the best validation process is to simply regenerate a valid cookie and compare it against the incoming cookie. In that case if the incoming cookie fails the check, a valid cookie is readily available to be sent to the client.

The server MAY return a cookie request option, e.g., a null cookie, to signal the support of Fast Open without generating cookies, for probing or debugging purposes.

4.1.3. Client Cookie Handling

The client MUST cache cookies from servers for later Fast Open connections. For a multi-homed client, the cookies are both client and server IP dependent. Beside the cookie, we RECOMMEND that the client caches the MSS and RTT to the server to enhance performance.

The MSS advertised by the server is stored in the cache to determine the maximum amount of data that can be supported in the SYN packet. This information is needed because data is sent before the server announces its MSS in the SYN-ACK packet. Without this information, the data size in the SYN packet is limited to the default MSS of 536 bytes [RFC1122]. The client SHOULD update the cache MSS value whenever it discovers new MSS value, e.g., through path MTU discovery.

Caching RTT allows seeding a more accurate SYN timeout than the default value [RFC6298]. This lowers the performance penalty if the network or the server drops the SYN packets with data or the cookie options (See "Reliability and Deployment Issues" section below).

The cache replacement algorithm is not specified and is left for the implementations.

Note that before TFO sees wide deployment, clients are advised to also cache negative responses from servers in order to reduce the amount of futile TFO attempts. Since TFO is enabled on a per-service port basis but cookies are independent of service ports, clients'

cache should include remote port numbers too.

4.2. Fast Open Protocol

One predominant requirement of TFO is to be fully compatible with existing TCP implementations, both on the client and the server sides.

The server keeps two variables per listening port:

FastOpenEnabled: default is off. It MUST be turned on explicitly by the application. When this flag is off, the server does not perform any TFO related operations and MUST ignore all cookie options.

PendingFastOpenRequests: tracks number of TFO connections in SYN-RCVD state. If this variable goes over a preset system limit, the server SHOULD disable TFO for all new connection requests until PendingFastOpenRequests drops below the system limit. This variable is used for defending some vulnerabilities discussed in the "Security Considerations" section.

The server keeps a FastOpened flag per TCB to mark if a connection has successfully performed a TFO.

4.2.1. Fast Open Cookie Request

Any client attempting TFO MUST first request a cookie from the server with the following steps:

1. The client sends a SYN packet with a Fast Open Cookie Request option.

2. The server SHOULD respond with a SYN-ACK based on the procedures in the "Server Cookie Handling" section. This SYN-ACK SHOULD contain a Fast Open Cookie option if the server currently supports TFO for this listener port.

3. If the SYN-ACK contains a Fast Open Cookie option, the client replaces the cookie and other information as described in the "Client Cookie Handling" section. Otherwise, if the SYN-ACK is first seen, i.e.,not a (spurious) retransmission, the client MAY remove the server information from the cookie cache. If the SYN-ACK is a spurious retransmission without valid Fast Open Cookie Option, the client does nothing to the cookie cache for the reasons below.

The network or servers may drop the SYN or SYN-ACK packets with the new cookie options which causes SYN or SYN-ACK timeouts. We RECOMMEND

both the client and the server retransmit SYN and SYN-ACK without the cookie options on timeouts. This ensures the connections of cookie requests will go through and lowers the latency penalties (of dropped SYN/SYN-ACK packets). The obvious downside for maximum compatibility is that any regular SYN drop will fail the cookie (although one can argue the delay in the data transmission till after 3WHS is justified if the SYN drop is due to network congestion). Next section describes a heuristic to detect such drops when the client receives the SYN-ACK.

We also RECOMMEND the client to record servers that failed to respond to cookie requests and only attempt another cookie request after certain period. An alternate proposal is to request cookie in FIN instead since FIN-drop by incompatible middle-box does not affect latency. However such paths are likely to drop SYN packet with data later, and many applications close the connections with RST instead, so the actual benefit of this approach is not clear.

4.2.2. TCP Fast Open

Once the client obtains the cookie from the target server, the client can perform subsequent TFO connections until the cookie is expired by the server. The nature of TCP sequencing makes the TFO specific changes relatively small in addition to [RFC793].

Client: Sending SYN

To open a TFO connection, the client MUST have obtained the cookie from the server:

1. Send a SYN packet.

   a. If the SYN packet does not have enough option space for the Fast Open Cookie option, abort TFO and fall back to regular 3WHS.

   b. Otherwise, include the Fast Open Cookie option with the cookie of the server. Include any data up to the cached server MSS or default 536 bytes.

2. Advance to SYN-SENT state and update SND.NXT to include the data accordingly.

3. If RTT is available from the cache, seed SYN timer according to [RFC6298].

To deal with network or servers dropping SYN packets with payload or unknown options, when the SYN timer fires, the client SHOULD retransmit a SYN packet without data and Fast Open Cookie options.

Server: Receiving SYN and responding with SYN-ACK

Upon receiving the SYN packet with Fast Open Cookie option:

1. Initialize and reset a local FastOpened flag. If FastOpenEnabled is false, go to step 5.

2. If PendingFastOpenRequests is over the system limit, go to step 5.

3. If IsCookieValid() in section 4.1.2 returns false, go to step 5.

4. Buffer the data and notify the application. Set FastOpened flag and increment PendingFastOpenRequests.

5. Send the SYN-ACK packet. The packet MAY include a Fast Open Option. If FastOpened flag is set, the packet acknowledges the SYN and data sequence. Otherwise it acknowledges only the SYN sequence. The server MAY include data in the SYN-ACK packet if the response data is readily available. Some application may favor delaying the SYN-ACK, allowing the application to process the request in order to produce a response, but this is left to the implementation.

6. Advance to the SYN-RCVD state. If the FastOpened flag is set, the server MUST follow the congestion control [RFC5681], in particular the initial congestion window [RFC3390], to send more data packets.

If the SYN-ACK timer fires, the server SHOULD retransmit a SYN-ACK segment with neither data nor Fast Open Cookie options for compatibility reasons.

Client: Receiving SYN-ACK

The client SHOULD perform the following steps upon receiving the SYN-ACK:

1. Update the cookie cache if the SYN-ACK has a Fast Open Cookie Option or MSS option or both.

2. Send an ACK packet. Set acknowledgment number to RCV.NXT and include the data after SND.UNA if data is available.

3. Advance to the ESTABLISHED state.

Note there is no latency penalty if the server does not acknowledge the data in the original SYN packet. The client SHOULD retransmit any unacknowledged data in the first ACK packet in step 2. The data exchange will start after the handshake like a regular TCP

connection.

If the client has timed out and retransmitted only regular SYN packets, it can heuristically detect paths that intentionally drop SYN with Fast Open option or data. If the SYN-ACK acknowledges only the initial sequence and does not carry a Fast Open cookie option, presumably it is triggered by a retransmitted (regular) SYN and the original SYN or the corresponding SYN-ACK was lost.

Server: Receiving ACK

Upon receiving an ACK acknowledging the SYN sequence, the server decrements PendingFastOpenRequests and advances to the ESTABLISHED state. No special handling is required further.

5. Reliability and Deployment Issues

Network or Hosts Dropping SYN packets with data or unknown options

A study [MAF04] found that some middle-boxes and end-hosts may drop packets with unknown TCP options incorrectly. Studies [LANGLEY06, HNRGHT11] both found that 6% of the probed paths on the Internet drop SYN packets with data or with unknown TCP options. The TFO protocol deals with this problem by retransmitting SYN without data or cookie options and we recommend tracking these servers in the client.

Server Farms

A common server-farm setup is to have many physical hosts behind a load-balancer sharing the same server IP. The load-balancer forwards new TCP connections to different physical hosts based on certain load-balancing algorithms. For TFO to work, the physical hosts need to share the same key and update the key at about the same time.

Network Address Translation (NAT)

The hosts behind NAT sharing same IP address will get the same cookie to the same server. This will not prevent TFO from working. But on some carrier-grade NAT configurations where every new TCP connection from the same physical host uses a different public IP address, TFO does not provide latency benefit. However, there is no performance penalty either as described in Section "Client: Receiving SYN-ACK".

6. Security Considerations

The Fast Open cookie stops an attacker from trivially flooding spoofed SYN packets with data to burn server resources or to mount an

amplified reflection attack on random hosts. The server can defend against spoofed SYN floods with invalid cookies using existing techniques [RFC4987]. We note that generating bogus cookies is usually cheaper than validating them. But the additional cost of validating the cookies, inherent to any authentication scheme, may not be substantial compared to processing a regular SYN packet.

However, the attacker may still obtain cookies from some compromised hosts, then flood spoofed SYN with data and "valid" cookies (from these hosts or other vantage points). With DHCP, it's possible to obtain cookies of past IP addresses without compromising any host. Below we identify new vulnerabilities of TFO and describe the countermeasures.

6.1. Server Resource Exhaustion Attack by SYN Flood with Valid Cookies

Like regular TCP handshakes, TFO is vulnerable to such an attack. But the potential damage can be much more severe. Besides causing temporary disruption to service ports under attack, it may exhaust server CPU and memory resources.

For this reason it is crucial for the TFO server to limit the maximum number of total pending TFO connection requests, i.e., PendingFastOpenRequests. When the limit is exceeded, the server temporarily disables TFO entirely as described in "Server Cookie Handling". Then subsequent TFO requests will be downgraded to regular connection requests, i.e., with the data dropped and only SYN acknowledged. This allows regular SYN flood defense techniques [RFC4987] like SYN-cookies to kick in and prevent further service disruption.

There are other subtle but important differences in the vulnerability between TFO and regular TCP handshake. Before the SYN flood attack broke out in the late '90s, typical listener's max qlen was small, enough to sustain the highest expected new connection rate and the average RTT for the SYN-ACK packets to be acknowledged in time. E.g., if a server is designed to handle at most 100 connection requests per second, and the average RTT is 100ms, a max qlen on the order of 10 will be sufficient.

This small max qlen made it very easy for any attacker, even equipped with just a dailup modem to the Internet, to cause major disruptions to a web site by simply throwing a handful of "SYN bombs" at its victim of choice. But for this attack scheme to work, the attacker must pick a non-responsive source IP address to spoof with. Otherwise the SYN-ACK packet will trigger TCP RST from the host whose IP address has been spoofed, causing corresponding connection to be removed from the server's listener queue hence defeating the attack.

In other words, the main damage of SYN bombs against the standard TCP stack is not directly from the bombs themselves costing TCP processing overhead or host memory, but rather from the spoofed SYN packets filling up the often small listener's queue.

On the other hand, TFO SYN bombs can cause damage directly if admitted without limit into the stack. The RST packets from the spoofed host will fuel rather than defeat the SYN bombs as compared to the non-TFO case, because the attacker can flood more SYNs with data to cost more data processing resources. For this reason, a TFO server needs to monitor the connections in SYN-RCVD being reset in addition to imposing a reasonable max qlen. Implementations may combine the two, e.g., by continuing to account for those connection requests that have just been reset against the listener's PendingFastOpenRequests until a timeout period has passed.

Limiting the maximum number of pending TFO connection requests does make it easy for an attacker to overflow the queue, causing TFO to be disabled. We argue that causing TFO to be disabled is unlikely to be of interest to attackers because the service will remain intact without TFO hence there is hardly any real damage.

6.2. Amplified Reflection Attack to Random Host

Limiting PendingFastOpenRequests with a system limit can be done without Fast Open Cookies and would protect the server from resource exhaustion. It would also limit how much damage an attacker can cause through an amplified reflection attack from that server. However, it would still be vulnerable to an amplified reflection attack from a large number of servers. An attacker can easily cause damage by tricking many servers to respond with data packets at once to any spoofed victim IP address of choice.

With the use of Fast Open Cookies, the attacker would first have to steal a valid cookie from its target victim. This likely requires the attacker to compromise the victim host or network first.

The attacker here has little interest in mounting an attack on the victim host that has already been compromised. But she may be motivated to disrupt the victim's network. Since a stolen cookie is only valid for a single server, she has to steal valid cookies from a large number of servers and use them before they expire to cause sufficient damage without triggering the defense in the previous section.

One can argue that if the attacker has compromised the target network or hosts, she could perform a similar but simpler attack by injecting bits directly. The degree of damage will be identical, but TFO-

specific attack allows the attacker to remain anonymous and disguises the attack as from other servers.

The best defense is for the server not to respond with data until handshake finishes. In this case the risk of amplification reflection attack is completely eliminated. But the potential latency saving from TFO may diminish if the server application produces responses earlier before the handshake completes.

6.3 Attacks from behind sharing public IPs (NATs)

An attacker behind NAT can easily obtain valid cookies to launch the above attack to hurt other clients that share the path. [BOB12] suggested that the server can extend cookie generation to include the TCP timestamp---GetCookie(IP_Address, Timestamp)---and implement it by encrypting the concatenation of the two values to generate the cookie. The client stores both the cookie and its corresponding timestamp, and echoes both in the SYN. The server then implements IsCookieValid(IP_Address, Timestamp, Cookie) by encrypting the IP and timestamp data and comparing it with the cookie value.

This enables the server to issue different cookies to clients that share the same IP address, hence can selectively discard those misused cookies from the attacker. However the attacker can simply repeat the attack with new cookies. The server would eventually need to throttle all requests from the IP address just like the current approach. Moreover this approach requires modifying [RFC 1323] to send non-zero Timestamp Echo Reply in SYN, potentially cause firewall issues. Therefore we believe the benefit may not outweigh the drawbacks.

7. Web Performance

7.1. HTTP persistent connection

TCP connection setup overhead has long been identified as a performance bottleneck for web applications [THK98]. HTTP persistent connection was proposed to mitigate this issue and has been widely deployed. However, [RCCJR11][AERG11] show that the average number of transactions per connection is between 2 and 4, based on large-scale measurements from both servers and clients. In these studies, the servers and clients both kept the idle connections up to several minutes, well into the human think time.

Can the utilization rate increase by keeping connections even longer? Unfortunately, this is problematic due to middle-boxes and rapidly growing mobile end hosts. One major issue is NAT. Studies

[HNESSK10][MQXMZ11] show that the majority of home routers and ISPs fail to meet the the 124 minutes idle timeout mandated in [RFC5382]. In [MQXMZ11], 35% of mobile ISPs timeout idle connections within 30 minutes. NAT boxes do not possess a reliable mechanism to notify end hosts when idle connections are removed from local tables, either due to resource constraints such as mapping table size, memory, or lookup overhead, or due to the limited port number and IP address space. Moreover, unmapped packets received by NAT boxes are often dropped silently. (TCP RST is not required by RFC5382.) The end host attempting to use these broken connections are often forced to wait for a lengthy TCP timeout. Thus the browser risks large performance penalty when keeping idle connections open. To circumvent this problem, some applications send frequent TCP keep-alive probes. However, this technique drains power on mobile devices [MQXMZ11]. In fact, power has become a prominent issue in modern LTE devices that mobile browsers close the HTTP connections within seconds or even immediately [SOUDERS11].

Idle connections also consume more memory resources. Due to the complexity of today's web applications, the application layer often needs orders of magnitude more memory than the TCP connection footprint. As a result, servers need to implement advanced resource management in order to support a large number of idle connections.

7.2 Case Study: Chrome Browser

[RCCJR11] studied Chrome browser performance based on 28 days of global statistics. Chrome browser keeps idle HTTP persistent connections up to 5 to 10 minutes. However the average number of the transactions per connection is only 3.3. Due to the low utilization, TCP 3WHS accounts up to 25% of the HTTP transaction network latency. The authors tested a Linux TFO implementation with TFO enabled Chrome browser on popular web sites in emulated environments such as residential broadband and mobile networks. They showed that TFO improves page load time by 10% to 40%. More detailed on the design tradeoffs and measurement can be found at [RCCJB11].

8. TFO's Applicability

TFO aims at latency conscious applications that are sensitive to TCP's initial connection setup delay. These application protocols often employ short-lived TCP connections, or employ long-lived connections but are more sensitive to the connection setup delay due to, e.g., a more strict connection fail-over requirement.

Only transaction-type applications where RTT constitutes a significant portion of the total end-to-end latency will likely benefit from TFO. Moreover, the client request must fit in the SYN

packet. Otherwise there may not be any saving in the total number of round trips required to complete a transaction.

To the extent possible applications protocols SHOULD employ long-lived connections to best take advantage of TCP's built-in congestion control algorithm, and to reduce the impact from TCP's connection setup overhead. E.g., for the web applications, P-HTTP will likely help and is much easier to deploy hence should be attempted first. TFO will likely provide further latency reduction on top of P-HTTP. But the additional benefit will depend on how much persistency one can get from HTTP in a given operating environment.

One alternative to short-lived TCP connection might be UDP, which is connectionless hence doesn't inflict any connection setup delay, and is best suited for application protocols that are transactional. Practical deployment issues such as middle-box and/or firewall traversal may severely limit the use of UDP based application protocols though.

Note that when the application employs too many short-lived connections, it may negatively impact network stability, as these connections often exit before TCP's congestion control algorithm kicks in. Implementations supporting large number of short-lived connections should employ temporal sharing of TCB data as described in [RFC2140].

More discussion on TCP Fast Open and its projected performance benefit can be found in [RCCJB11].

9. Related Work

9.1. T/TCP

TCP Extensions for Transactions [RFC1644] attempted to bypass the three-way handshake, among other things, hence shared the same goal but also the same set of issues as TFO. It focused most of its effort battling old or duplicate SYNs, but paid no attention to security vulnerabilities it introduced when bypassing 3WHS. Its TAO option and connection count, besides adding complexity, require the server to keep state per remote host, while still leaving it wide open for attacks. It is trivial for an attacker to fake a CC value that will pass the TAO test. Unfortunately, in the end its scheme is still not 100% bullet proof as pointed out by [PHRACK98].

As stated earlier, we take a practical approach to focus TFO on the security aspect, while allowing old, duplicate SYN packets with data after recognizing that 100% TCP semantics is likely infeasible. We

believe this approach strikes the right tradeoff, and makes TFO much simpler and more appealing to TCP implementers and users.

9.2. Common Defenses Against SYN Flood Attacks

TFO is still vulnerable to SYN flood attacks just like normal TCP handshakes, but the damage may be much worse, thus deserves a careful thought.

There have been plenty of studies on how to mitigate attacks from regular SYN flood, i.e., SYN without data [RFC4987]. But from the stateless SYN-cookies to the stateful SYN Cache, none can preserve data sent with SYN safely while still providing an effective defense.

The best defense may be to simply disable TFO when a host is suspected to be under a SYN flood attack, e.g., the SYN backlog is filled. Once TFO is disabled, normal SYN flood defenses can be applied. The "Security Consideration" section contains a thorough discussion on this topic.

9.3. TCP Cookie Transaction (TCPCT)

TCPCT [RFC6013] eliminates server state during initial handshake and defends spoofing DoS attacks. Like TFO, TCPCT allows SYN and SYN-ACK packets to carry data. However, TCPCT and TFO are designed for different goals and they are not compatible.

The TCPCT server does not keep any connection state during the handshake, therefore the server application needs to consume the data in SYN and (immediately) produce the data in SYN-ACK before sending SYN-ACK. Otherwise the application’s response has to wait until handshake completes. In contrary, TFO allows server to respond data during handshake. Therefore for many request-response style applications, TCPCT may not achieve same latency benefit as TFO.

Rapid-Restart [SIMPSON11] is based on TCPCT and shares similar goal as TFO. In Rapid-Restart, both the server and the client retain the TCP control blocks after a connection is terminated in order to allow/resume data exchange in next connection handshake. In contrary, TFO does not require keeping both TCB on both sides and is more scalable.

10. IANA Considerations

The Fast Open Cookie Option and Fast Open Cookie Request Option define no new namespace. The options require IANA allocate one value from the TCP option Kind namespace. Early implementation before the

allocation SHOULD follow [EXPOPT] and use experimental option 254 and magic number 0xF989 (16 bits), and migrate to the new option after the allocation according.

11. Acknowledgement

We thank Rick Jones, Bob Briscoe, Adam Langley, Matt Mathis, Neal Cardwell, Roberto Peon, and Tom Herbert for their feedbacks. We especially thank Barath Raghavan for his contribution on the security design of Fast Open.

12. References

12.1. Normative References

[RFC793] Postel, J. "Transmission Control Protocol", RFC 793, September 1981.

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, October 1989.

[RFC5382] S. Guha, Ed., Biswas, K., Ford B., Sivakumar S., Srisuresh, P., "NAT Behavioral Requirements for TCP", RFC 5382

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

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

12.2. Informative References

[AERG11] M. Al-Fares, K. Elmeleegy, B. Reed, and I. Gashinsky, "Overclocking the Yahoo! CDN for Faster Web Page Loads". In Proceedings of Internet Measurement Conference, November 2011.

[CDCM11] Chu, J., Dukkipati, N., Cheng, Y. and M. Mathis, "Increasing TCP's Initial Window", Internet-Draft draft-ietf-tcpm-initcwnd-02.txt (work in progress), October 2011.

[EXPOPT] Touch, Joe, "Shared Use of Experimental TCP Options", Internet-Draft draft-ietf-tcpm-experimental-options (work in progress), October 2012.

[HNESSK10] S. Haetoenen, A. Nyrhinen, L. Eggert, S. Strowes, P. Sarolahti, M. Kojo., "An Experimental Study of Home Gateway Characteristics". In Proceedings of Internet Measurement Conference. Octobor 2010

[HNRGHT11] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, H. Tokuda, "Is it Still Possible to Extend TCP?". In Proceedings of Internet Measurement Conference. November 2011.

[LANGLEY06] Langley, A, "Probing the viability of TCP extensions", URL http://www.imperialviolet.org/binary/ecntest.pdf

[MAF04] Medina, A., Allman, M., and S. Floyd, "Measuring

Interactions Between Transport Protocols and Middleboxes", In Proceedings of Internet Measurement Conference, October 2004.

[MQXMZ11] Z. Mao, Z. Qian, Q. Xu, Z. Mao, M. Zhang. "An Untold Story of Middleboxes in Cellular Networks", In Proceedings of SIGCOMM. August 2011.

[PHRACK98] "T/TCP vulnerabilities", Phrack Magazine, Volume 8, Issue 53 artical 6. July 8, 1998. URL http://www.phrack.com/issues.html?issue=53&id=6

[QWGMSS11] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, O. Spatscheck. "Profiling Resource Usage for Mobile Applications: A Cross-layer Approach", In Proceedings of International Conference on Mobile Systems. April 2011.

[RCCJB11] Radhakrishnan, S., Cheng, Y., Chu, J., Jain, A. and B. Raghavan, "TCP Fast Open". In Proceedings of 7th ACM CoNEXT Conference, December 2011.

[RFC1644] Braden, R., "T/TCP -- TCP Extensions for Transactions Functional Specification", RFC 1644, July 1994.

[RFC2140] Touch, J., "TCP Control Block Interdependence", RFC2140, April 1997.

[RFC4987] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", RFC 4987, August 2007.

[RFC6013] Simpson, W., "TCP Cookie Transactions (TCPCT)", RFC6013, January 2011.

[SIMPSON11] Simpson, W., "Tcp cookie transactions (tcpct) rapid restart", Internet draft draft-simpson-tcpct-rr-02.txt (work in progress), July 2011.

[SOUDERS11] S. Souders. "Making A Mobile Connection". http://www.stevesouders.com/blog/2011/09/21/making-a-mobile-connection/

[THK98] Touch, J., Heidemann, J., Obraczka, K., "Analysis of HTTP Performance", USC/ISI Research Report 98-463. December 1998.

[BOB12] Briscoe, B., "Some ideas building on draft-ietf-tcpm-fastopen-01", tcpm list, http://www.ietf.org/mail-archive/web/tcpm/current/

msg07192.html

Author's Addresses

Yuchung Cheng
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043, USA
EMail: ycheng@google.com

Jerry Chu
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043, USA
EMail: hkchu@google.com

Sivasankar Radhakrishnan
Department of Computer Science and Engineering
University of California, San Diego
9500 Gilman Dr
La Jolla, CA 92093-0404
EMail: sivasankar@cs.ucsd.edu

Arvind Jain
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043, USA
EMail: arvind@google.com

TCPM Working Group
Internet-Draft
Obsoletes: 2861 (if approved)
Updates: 5681 (if approved)
Intended status: Standards Track
Expires: August 18, 2013

G. Fairhurst
A. Sathiaseelan
University of Aberdeen
February 14, 2013

Updating TCP to support Rate-Limited Traffic
draft-ietf-tcpm-newcwv-00

Abstract

This document proposes an update to RFC 5681 to address issues that arise when TCP is used to support traffic that exhibits periods where the sending rate is limited by the application rather than the congestion window. It updates TCP to allow a TCP sender to restart quickly following either an idle or rate-limited interval. This method is expected to benefit applications that send rate-limited traffic using TCP, while also providing an appropriate response if congestion is experienced.

It also evaluates TCP Congestion Window Validation, CWV, an IETF experimental specification defined in RFC 2861, and concludes that CWV sought to address important issues, but failed to deliver a widely used solution. This document therefore proposes an update to the status of RFC 2861 by recommending it is moved from Experimental to Historic status, and that it is replaced by the current specification.

NOTE: The standards status of this WG document is under review for consideration as either Experimental (EXP) or Proposed Standard (PS). This decision will be made later as the document is finalised.

Status of this Memo

This Internet-Draft will expire on August 18, 2013.

Table of Contents

1. Introduction 4
2. Reviewing experience with TCP-CWV 4
3. Terminology 5
4. An updated TCP response to idle and application-limited periods 6
4.1. A method for preserving cwnd during the idle and application-limited periods. 7
4.2. The nonvalidated phase 7
4.3. TCP congestion control during the nonvalidated phase . . . 8
4.3.1. Response to congestion in the nonvalidated phase . . . 9
4.3.2. Adjustment at the end of the nonvalidated phase . . . 9
5. Determining a safe period to preserve cwnd 10
6. Security Considerations 11
7. IANA Considerations 11
8. Acknowledgments 11
9. Author Notes 12
9.1. Other related work 12
9.2. Revision notes 14
10. References 15
10.1. Normative References 15
10.2. Informative References 15
Authors' Addresses 16

1. Introduction

TCP is used to support a range of application behaviours. The TCP congestion window (cwnd) controls the number of unacknoeledged packets/bytes that a TCP flow may have in the network at any time, a value known as the FlightSize [RFC5681]. A bulk application will always have data available to transmit. The rate at which it sends is therefore limited by the maximum permitted by the receiver and congestion windows. In contrast, a rate-limited application will experience periods when the sender is either idle or is unable to send at the maximum rate permitted by the cwnd. This latter case is called rate-limited. The focus of this document is on the operation of TCP in such an idle or rate-limited case.

Standard TCP [RFC5681] requires the cwnd to be reset to the restart window (RW) when an application becomes idle. [RFC2861] noted that this TCP behaviour was not always observed in current implementations. Recent experiments [Bis08] confirm this to still be the case.

Standard TCP does not impose additional restrictions on the growth of the cwnd when a TCP sender is rate-limited. A rate-limited sender may therefore grow a cwnd far beyond that corresponding to the current transmit rate, resulting in a value that does not reflect current information about the state of the network path the flow is using. Use of such an invalid cwnd may result in reduced application performance and/or could significantly contribute to network congestion.

[RFC2861] proposed a solution to these issues in an experimental method known as Congestion Window Validation (CWV). CWV was intended to help reduce cases where TCP accumulated an invalid cwnd. The use and drawbacks of using CWV with an application are discussed in Section 2.

Section 3 defines relevant terminology.

Section 4 specifies an alternative to CWV that seeks to address the same issues, but does this in a way that is expected to mitigate the impact on an application that varies its sending rate. The method described applies to both a rate-limited and an idle condition.

2. Reviewing experience with TCP-CWV

RFC 2861 described a simple modification to the TCP congestion control algorithm that decayed the cwnd after the transition to a "sufficiently-long" idle period. This used the slow-start threshold

(ssthresh) to save information about the previous value of the congestion window. The approach relaxed the standard TCP behaviour [RFC5681] for an idle session, intended to improve application performance. CWV also modified the behaviour for a rate-limited session where a sender transmitted at a rate less than allowed by cwnd.

RFC 2861 has been implemented in some mainstream operating systems as the default behaviour [Bis08]. Analysis (e.g. [Bis10] [Fai12]) has shown that a TCP sender using CWV is able to use available capacity on a shared path after an idle period. This can benefit some applications, especially over long delay paths, when compared to the slow-start restart specified by standard TCP. However, CWV would only benefit an application if the idle period were less than several Retransmission Time Out (RTO) intervals [RFC6298], since the behaviour would otherwise be the same as for standard TCP, which resets the cwnd to the RTCP Restart Window (RW) after this period.

Experience with CWV suggests that although CWV benefits the network in a rate-limited scenario (reducing the probability of network congestion), the behaviour can be too conservative for many common rate-limited applications. This mechanism does not therefore offer the desirable increase in application performance for rate-limited applications and it is unclear whether applications actually use this mechanism in the general Internet.

It is therefore concluded that CWV is often a poor solution for many rate-limited applications. It has the correct motivation, but has the wrong approach to solving this problem.

3. 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].

The document assumes familiarity with the terminology of TCP congestion control [RFC5681].

The following new terminology is introduced:

Validated phase: The phase where the cwnd reflects a current estimate of the available path capacity.

Non-validated phase: The phase where the cwnd reflects a previous measurement of the available path capacity.

Non-validated period, NVP: The maximum period for which cwnd is preserved in the non-validated phase.

Rate-limited: A TCP flow that does not consume more than one half of cwnd, and hence operates in the non-validated phase.

pipe ACK: The measured volume of data that was acknowledged by the network per RTT.

4. An updated TCP response to idle and application-limited periods

This section proposes an update to the TCP congestion control behaviour during an idle or rate-limited period. The new method permits a TCP sender to preserve the cwnd when an application becomes idle for a period of time (to be known as the non-validated period, NVP, see section 5). The period where actual usage is less than allowed by cwnd, is named as the non-validated phase. This method allows an application to resume transmission at a previous rate without incurring the delay of slow-start. However, if the TCP sender experiences congestion using the preserved cwnd, it is required to immediately reset the cwnd to an appropriate value specified by the method. If a sender does not take advantage of the preserved cwnd within the NVP, the value of cwnd is reduced, ensuring the value better reflects the capacity that was recently actually used.

The method requires that the TCP SACK option [RFC3517]is enabled. This allows the sender to select an appropriate value for the cwnd following a congestion event that is based on the measured path capacity, and better reflects the fair-share. A similar approach was proposed by TCP Jump Start [Liu07], as a congestion response after more rapid opening of a TCP connection.

It is expected that this update will satisfy the requirements of many rate-limited applications and at the same time provide an appropriate method for use in the Internet. It also reduces the incentive for an application to send data simply to keep transport congestion state. (This is sometimes known as "padding").

The new method does not differentiate between times when the sender has become idle or rate-limited. This is partly a response to recognition that some applications wish to transmit at a rate less than allowed by the sender cwnd, and that it can be hard to make a distinction between rate-limited and idle behaviour. This is expected to encourage applications and TCP stacks to use standards-based congestion control methods. It may also encourage the use of long-lived connections where this offers benefit (such as persistent

http).

The method is specified in following subsections.

4.1. A method for preserving cwnd during the idle and application-limited periods.

The method described in this document updates [RFC5681]. Use of the method REQUIRES a TCP sender and the corresponding receiver to enable the TCP SACK option [RFC3517].

[RFC5681] defines a variable, FlightSize, that indicates the amount of outstanding data in the network. This is assumed to be equal to the value of Pipe calculated based on the pipe algorithm [RFC3517]. In RFC5681 this value is used during loss recovery, whereas in this method a new variable "pipeACK" is introduced and used to determine if the sender has validated the cwnd.

The value of pipeACK is initialised to the maxium value. This value is used to inhibt entering the nonvalidated phase until the first measurement of pipeACK completes.

A sender is not required to continuously track the pipeACK value, but MUST set this variable to the volume of data that was acknowledged by the network per measured Round Trip Time (RTT), with a sampling period of not less than one measurement for Min(RTT, 1 second). Using the variables defined in [RFC3517]. This could be implemented by caching the value of HighACK and after one RTT assigning pipeACK to the difference between the cached HighACK value and the current HighACK value. Other equivalent methods may be used.

4.2. The nonvalidated phase

The updated method creates a new TCP sender phase that captures whether the cwnd reflects a validated or non-validated value. The phases are defined as:

- o Validated phase: pipeACK >=(1/2)*cwnd. This is the normal phase, where cwnd is expected to be an approximate indication of the available capacity currently available along the network path, and the standard methods are used to increase cwnd (currently [RFC5681]). The rule for transitioning to the non-validated phase is specified in section 4.3.

- o Non-validated phase: pipeACK <(1/2)*cwnd. This is the phase where the cwnd has a value based on a previous measurement of the available capacity, and the usage of this capacity has not been validated in the previous RTT. That is, when it is not known

whether the cwnd reflects the currently available capacity along the network path. The mechanisms to be used in this phase seek to determine a safe value for cwnd and an appropriate reaction to congestion. These mechanisms are specified in section 4.3.

A sender starts a TCP connection in the Validated phase.

The value 1/2 was selected to reduce the effects of variations in the measured pipeACK, and to allow the sender some flexibility in when it sends data.

4.3. TCP congestion control during the nonvalidated phase

A TCP sender MUST enter the non-validated phase when the measured pipeACK is less than (1/2)*cwnd.

A TCP sender that enters the non-validated phase will preserve the cwnd (i.e., this neither grows nor reduces while the sender remains in this phase). The phase is concluded after a fixed period of time (the NVP, as explained in section 4.3.2) or when the sender transmits sufficient data so that pipeACK > (1/2)*cwnd (i.e. it is no longer rate-limited).

The behaviour in the non-validated phase is specified as:

- The cwnd is not increased when ACK packets are received in this phase.

- If the sender receives an indication of congestion while in the non-validated phase (i.e. detects loss, or an Explicit Congestion Notification, ECN, mark [RFC3168]), the sender MUST exit the non-validated phase (reducing the cwnd as defined in section 4.3.1).

- If the Retransmission Time Out (RTO) expires while in the non-validated phase, the sender MUST exit the non-validated phase. It then resumes using the Standard TCP RTO mechanism [RFC5681]. (The resulting reduction of cwnd described in section 4.3.2 is appropriate, since any accumulated path history is considered unreliable).

- A sender that measures a pipeACK greater than (1/2)*cwnd SHOULD enter the validated phase. (A rate-limited sender will not normally be impacted by whether it is in a validated or non-validate phase, since it will normally not consume the entire cwnd. However a change to the validated phase will release the sender from constraints on the growth of cwnd, and restore the use of the standard congestion response.)

4.3.1. Response to congestion in the nonvalidated phase

Reception of congestion feedback while in the non-validated phase is interpreted as an indication that it was inappropriate for the sender to use the preserved cwnd. The sender is therefore required to quickly reduce the rate to avoid further congestion. Since the cwnd does not have a validated value, a new cwnd value must be selected based on the utilised rate.

A sender that detects a packet-drop or receives an ECN marked packet MUST calculate a safe cwnd, by setting it to the value specified in Section 3.2 of [RFC5681].

At the end of the recovery phase, the TCP sender MUST reset the cwnd using the method below:

```
        cwnd = ((FlightSize - R)/2).
```

Where, R is the volume of data that was reported as unacknowledged by the SACK information. This follows the method proposed for Jump Start [Liu07].

The inclusion of the term R makes this adjustment more conservative than standard TCP. (This is required, since the sender may have sent more segments than a Standard TCP sender would have done. The additional reduction is beneficial when the FlightSize significantly overshoots the available path capacity incurring significant loss, for instance an intense traffic burst following a non-validated period.)

If the sender implements a method that allows it to identify the number of ECN-marked segments within a window that were observed by the receiver, the sender SHOULD use the method above, further reducing R by the number of marked segments.

The sender MUST also re-initialise the pipeACK variable to the maxium value. This ensures that standard TCP methods are used immediately after completing loss recovery.

4.3.2. Adjustment at the end of the nonvalidated phase

During the non-validated phase, a sender can produce bursts of data of up to the cwnd in size. While this is no different to standard TCP, it is desirable to control the maximum burst size, e.g. by setting a burst size limit, using a pacing algorithm, or some other method [Hug01].

An application that remains in the non-validated phase for a period greater than the NVP is required to adjust its congestion control

state. If the sender exits the non-validated phase after this period, it MUST update the ssthresh:

ssthresh = max(ssthresh, 3*cwnd/4).

(This adjustment of ssthresh ensures that the sender records that it has safely sustained the present rate. The change is beneficial to rate-limited flows that encounter occasional congestion, and could otherwise suffer an unwanted additional delay in recovering the sending rate.)

The sender MUST then update cwnd to be not greater than:

cwnd = max(1/2*cwnd, IW).

Where IW is the TCP inital window [RFC5681].

(This adjustment ensures that sender responds conservatively at the end of the non-validated phase by reducing the cwnd to better reflect the current sending rate of the sender. The cwnd update does not take into account FlightSize or pipeACK because these values only reflect data during the last RTT and do not reflect the average or peak sending rate.)

After completing this adjustment, the sender MAY re-enter the non-validated phase, if required (see section 4.2).

5. Determining a safe period to preserve cwnd

This section documents the rationale for selecting the maximum period that cwnd may be preserved, known as the non-validated period, NVP.

Limiting the period that cwnd may be preserved avoids undesirable side effects that would result if the cwnd were to be kept unecessarily high for an arbitrary long period, which was a part of the problem that CWV originally attempted to address. The period a sender may safely preserve the cwnd, is a function of the period that a network path is expected to sustain the capacity reflected by cwnd. There is no ideal choice for this time.

A period of five minutes was chosen for this NVP. This is a compromise that was larger than the idle intervals of common applications, but not sufficiently larger than the period for which the capacity of an Internet path may commonly be regarded as stable. The capacity of wired networks is usually relatively stable for periods of several minutes and that load stability increases with the

capacity. This suggests that cwnd may be preserved for at least a few minutes.

There are cases where the TCP throughput exhibits significant variability over a time less than five minutes. Examples could include wireless topologies, where TCP rate variations may fluctuate on the order of a few seconds as a consequence of medium access protocol instabilities. Mobility changes may also impact TCP performance over short time scales. Senders that observe such rapid changes in the path characteristic may also experience increased congestion with the new method, however such variation would likely also impact TCP's behaviour when supporting interactive and bulk applications.

Routing algorithms may modify the network path, disrupting the RTT measurement and changing the capacity available to a TCP connection, however such changes do not often occur within a time frame of a few minutes.

The value of five minutes is therefore expected to be sufficient for most current applications. Simulation studies (e.g. [Bis11]) also suggest that for many practical applications, the performance using this value will not be significantly different to that observed using a non-standard method that does not reset the cwnd after idle.

Finally, other TCP sender mechanisms have used a 5 minute timer, and there could be simplifications in some implementations by reusing the same interval. TCP defines a default user timeout of 5 minutes [RFC0793] i.e. how long transmitted data may remain unacknowledged before a connection is forcefully closed.

6. Security Considerations

General security considerations concerning TCP congestion control are discussed in [RFC5681]. This document describes an algorithm that updates one aspect of the congestion control procedures, and so the considerations described in RFC 5681 also apply to this algorithm.

7. IANA Considerations

There are no IANA considerations.

8. Acknowledgments

The authors acknowledge the contributions of Dr I Biswas and Dr R

Secchi in supporting the evaluation of CWV and for their help in developing the mechanisms proposed in this draft. We also acknowledge comments received from the Internet Congestion Control Research Group, in particular Yuchung Cheng, Mirja Kuehlewind, and Joe Touch.

9. Author Notes

9.1. Other related work

There are several issues to be discussed more widely:

o Should the method explicitly state a procedure for limiting burstiness or pacing?

This is often regarded as good practice, but is not presently a formal part of TCP. draft-hughes-restart-00.txt provides some discussion of this topic.

o There are potential interactions with the proposal to raise the TCP initial Window to ten segments, do these cases need to be elaborated?

This relates to draft-ietf-tcpm-initcwnd.

The two methods have different functions and different response to loss/congestion.

IW=10 proposes an experimental update to TCP that would allow faster opening of the cwnd, and also a large (same size) restart window. This approach is based on the assumption that many forward paths can sustain bursts of up to ten segments without (appreciable) loss. Such a significant increase in cwnd must be matched with an equally large reduction of cwnd if loss/congestion is detected, and such a congestion indication is likely to require future use of IW=10 to be disabled for this path for some time. This guards against the unwanted behaviour of a series of short flows continuously flooding a network path without network congestion feedback.

In contrast, new-CWV proposes a standards-track update with a rationale that relies on recent previous path history to select an appropriate cwnd after restart.

The behaviour differs in three ways:

1) For applications that send little initially, new-cwv may constrain more than IW=10, but would not require the connection to reset any path information when a restart incurred loss. In contrast, new-cwv would allow the TCP connection to preserve the cached cwnd, any loss, would impact cwnd, but not impact other flows.

2) For applications that utilise more capacity than provided by a cwnd=10, this method would permit a larger restart window compared to a restart using IW=10. This is justified by the recent path history.

3) new-CWV is attended to also be used for rate-limited applications, where the application sends, but does not seek to fully utilise the cwnd. In this case, new-cwv constrains the cwnd to that justified by the recent path history. The performance trade-offs are hence different, and it would be possible to enable new-cwv when also using IW=10, and yield the benefits of this.

o There is potential overlap with the Laminar proposal (draft-mathis-tcpm-tcp-laminar)

The current draft was intended as a standards-track update to TCP, rather than a new transport variant. At least, it would be good to understand how the two interact and whether there is a possibility of a single method.

o There is potential performance loss in loss of a short burst (off list with M Allman)

A sender can transmit several segments then become idle. If the first segments are all ACK'ed the ssthresh collapses to a small value (no new data is sent by the idle sender). Loss of the later data results in congestion (e.g. maybe a RED drop or some other cause, rather than the peak rate of this flow). When performs loss recovery it may have an appreciable pipeACK and cwnd, but a very low flight size - the Standard algorithm results in an unusually low cwnd (1/2 Flight size).

A constant rate flow would have maintained a flight size appropriate to pipeACK (cwnd if it is a bulk flow).

This could be fixed by adding a new state variable? It could also be argued this is a corner case (e.g. loss of only the last segments would have resulted in RTO), the impact could be significant.

9.2. Revision notes

RFC-Editor note: please remove this section prior to publication.

Draft 03 was submitted to ICCRG to receive comments and feedback.

Draft 04 contained the first set of clarifications after feedback:

- o Changed name to application limited and used the term rate-limited in all places.
- o Added justification and many minor changes suggested on the list.
- o Added text to tie-in with more accurate ECN marking.
- o Added ref to Hug01

Draft 05 contained various updates:

- o New text to redefine how to measure the acknowledged pipe, differentiating this from the FlightSize, and hence avoiding previous issues with infrequent large bursts of data not being validated. A key point new feature is that pipeACK only triggers leaving the NVP after the size of the pipe has been acknowledged. This removed the need for hysteresis.
- o Reduction values were changed to 1/2, following analysis of suggestions from ICCRG. This also sets the "target" cwnd as twice the used rate for non-validated case.
- o Introduced a symbolic name (NVP) to denote the 5 minute period.

Draft 06 contained various updates:

- o Required reset of pipeACK after congestion.
- o Added comment on the effect of congestion after a short burst (M. Allman).
- o Correction of minor Typos.

WG draft 01 contained various updates:

- o Updaed initialisation of pipeACK to maximum value.
- o Added note on intended status still to be determined.

10. References

10.1. Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2861] Handley, M., Padhye, J., and S. Floyd, "TCP Congestion Window Validation", RFC 2861, June 2000.

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, September 2001.

[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.

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

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

10.2. Informative References

[Bis08] Biswas and Fairhurst, "A Practical Evaluation of Congestion Window Validation Behaviour, 9th Annual Postgraduate Symposium in the Convergence of Telecommunications, Networking and Broadcasting (PGNet), Liverpool, UK", June 2008.

[Bis10] Biswas, Sathiaseelan, Secchi, and Fairhurst, "Analysing TCP for Bursty Traffic, Int'l J. of Communications, Network and System Sciences, 7(3)", June 2010.

[Bis11] Biswas, "PhD Thesis, Internet congestion control for variable rate TCP traffic, School of Engineering, University of Aberdeen", June 2011.

[Fai12] Fairhurst, Biswas, Biswas, and Biswas, "Enhancing TCP Performance to support Variable-Rate Traffic, 2nd Capacity Sharing Workshop, ACM CoNEXT, Nice, France, 10th December 2012.", June 2008.

[Hug01] Hughes, Touch, and Heidemann, "√√Issues in TCP Slow-Start Restart After Idle (Work-in-Progress)", December 2001.

[Liu07] Liu, Allman, Jiny, and Wang, "Congestion Control without a Startup Phase, 5th International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet), Los Angeles, California, USA", February 2007.

Authors' Addresses

Godred Fairhurst
University of Aberdeen
School of Engineering
Fraser Noble Building
Aberdeen, Scotland AB24 3UE
UK

Email: gorry@erg.abdn.ac.uk
URI: http://www.erg.abdn.ac.uk

Arjuna Sathiaseelan
University of Aberdeen
School of Engineering
Fraser Noble Building
Aberdeen, Scotland AB24 3UE
UK

Email: arjuna@erg.abdn.ac.uk
URI: http://www.erg.abdn.ac.uk