

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

N. Dukkupati
N. Cardwell
Y. Cheng
M. Mathis
Google, Inc
February 25, 2013

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

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

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

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

This Internet-Draft will expire on August 29, 2013.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

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 $\max(2 * \text{SRTT}, 10\text{ms})$, 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 $\max(2 \cdot \text{SRTT}, 10\text{ms})$.

FlightSize == 1: schedule PTO in $\max(2 \cdot \text{SRTT}, 1.5 \cdot \text{SRTT} + \text{WCDelAckT})$.

If RTO is earlier, schedule PTO in its place: $\text{PTO} = \min(\text{RTO}, \text{PTO})$.

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 $2 \times \text{SRTT}$ 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	FAK fast recovery	all repaired
(5)	>=5 L	..LS	FAK 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. $(\text{timeouts_tlp} - \text{timeouts_baseline}) / \text{timeouts_baseline} = 15\%$. 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. $(\text{number of probe segments} / \text{number of outgoing segments}) * 100 = 0.48\%$. 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., Dukkkipati, 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

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

G. Fairhurst
A. Sathiaselalan
University of Aberdeen
September 14, 2012

Updating TCP to support Rate-Limited Traffic
draft-fairhurst-tcpm-newcwv-05

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.

Status of this Memo

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

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

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

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

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
2. Reviewing experience with TCP-CWV	3
3. Terminology	4
4. An updated TCP response to idle and application-limited periods	5
4.1. A method for preserving cwnd during the idle and application-limited periods.	6
4.2. The nonvalidated phase	6
4.3. TCP congestion control during the nonvalidated phase	7
4.3.1. Response to congestion in the nonvalidated phase	7
4.3.2. Adjustment at the end of the nonvalidated phase	8
5. Determining a safe period to preserve cwnd	9
6. Security Considerations	10
7. IANA Considerations	10
8. Acknowledgments	10
9. Author Notes	10
9.1. Other related work	11
9.2. Revision notes	12
10. References	13
10.1. Normative References	13
10.2. Informative References	13
Authors' Addresses	14

1. Introduction

TCP is used to support a range of application behaviours. The TCP congestion window (cwnd) controls the number of packets/bytes that a TCP flow may have in the network at any time. The unacknowledged volume of data that a TCP flow has in the network at a specific time is 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 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]) 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). This 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 is enabled. This allows the sender to select a cwnd following a congestion event that is based on the measured path capacity, better reflecting 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.

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: $\text{pipeACK} \geq (1/2) * \text{cwnd}$. 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: $\text{pipeACK} < (1/2) * \text{cwnd}$. 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 values $1/2$ 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:

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

$$\text{cwnd} = ((\text{FlightSize} - R)/2).$$

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.

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:

$$\text{ssthresh} = \max(\text{ssthresh}, 3 * \text{cwnd} / 4).$$

(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:

$$\text{cwnd} = \max(1/2 * \text{cwnd}, \text{IW}).$$

Where IW is the TCP initial 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 as these values only reflect the last RTT worth of data and do not reflect the average of 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.

Preserving cwnd avoids undesirable side effects that would result if the cwnd were to be preserved 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 as 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 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 interaction 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 intended 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.

9.2. Revision notes

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.

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, Sathiaselalan, Secchi, and Fairhurst, "Analysing TCP for Bursty Traffic, Int'l J. of Communications, Network and System Sciences, 7(3)", June 2010.
- [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 Sathiasseelan
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>

TCP Maintenance and Minor Extensions
(tcpm)
Internet-Draft
Intended status: Experimental
Expires: April 25, 2013

P. Hurtig
Karlstad University
A. Petlund
Simula Research Laboratory AS
M. Welzl
University of Oslo
October 22, 2012

TCP and SCTP RTO Restart
draft-hurtig-tcpm-rtorestart-03

Abstract

This document describes a modified algorithm for managing the TCP and SCTP retransmission timers that provides faster loss recovery when a connection's amount of outstanding data is small. The modification allows the transport to restart its retransmission timer more aggressively in situations where fast retransmit cannot be used. This enables faster loss detection and recovery for connections that are short-lived or application-limited.

Status of this Memo

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

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

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

This Internet-Draft will expire on April 25, 2013.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents

carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

1. Introduction

TCP uses two mechanisms to detect segment loss. First, if a segment is not acknowledged within a certain amount of time, a retransmission timeout (RTO) occurs, and the segment is retransmitted [RFC6298]. While the RTO is based on measured round-trip times (RTTs) between the sender and receiver, it also has a conservative lower bound of 1 second to ensure that delayed segments are not mistaken as lost. Second, when a sender receives duplicate acknowledgments, the fast retransmit algorithm infers segment loss and triggers a retransmission. Duplicate acknowledgments are generated by a receiver when out-of-order segments arrive. As both segment loss and segment reordering cause out-of-order arrival, fast retransmit waits for three duplicate acknowledgments before considering the segment as lost. In some situations, however, the number of outstanding segments is not enough to trigger three duplicate acknowledgments, and the sender must rely on lengthy RTOs for loss recovery.

The amount of outstanding segments can be small for several reasons:

- (1) The connection is limited by the congestion control when the path has a low total capacity (bandwidth-delay product) or the connection's share of the capacity is small. It is also limited by the congestion control in the first RTTs of a connection or after an RTO when the available capacity is probed using slow-start.
- (2) The connection is limited by the receiver's available buffer space.
- (3) The connection is limited by the application if the available capacity of the path is not fully utilized (e.g. interactive applications), or at the end of a transfer, which is frequent if the total amount of data is small (e.g. web traffic).

The first two situations can occur for any flow, as external factors at the network and/or host level cause them. The third situation primarily affects flows that are short or have a low transmission rate. Typical examples of applications that produce short flows are web servers. [RJ10] shows that 70% of all web objects, found at the top 500 sites, are too small for fast retransmit to work. [BPS98]

shows that about 56% of all retransmissions sent by a busy web server are sent after RTO expiry. While the experiments were not conducted using SACK [RFC2018], only 4% of the RTO-based retransmissions could have been avoided. Applications have a low transmission rate when data is sent in response to actions, or as a reaction to real life events. Typical examples of such applications are stock trading systems, remote computer operations and online games. What is special about this class of applications is that they are time-dependant, and extra latency can reduce the application service level [P09]. Although such applications may represent a small amount of data sent on the network, a considerable number of flows have such properties and the importance of low latency is high.

The RTO restart approach outlined in this document makes the RTO slightly more aggressive when the number of outstanding segments is small, in an attempt to enable faster loss recovery for all segments while being robust to reordering. While it still conforms to the requirement in [RFC6298] that segments must not be retransmitted earlier than RTO seconds after their original transmission, it could increase the chance for a spurious timeout, which could degrade performance when the congestion window (cwnd) is large -- for example, when an application sends enough data to reach a cwnd covering 100 segments and then stops. The likelihood and potential impact of this problem as well as possible mitigation strategies are currently under investigation.

While this document focuses on TCP, the described changes are also valid for the Stream Control Transmission Protocol (SCTP) [RFC4960] which has similar loss recovery and congestion control algorithms.

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].

2. RTO Restart Overview

The RTO management algorithm described in [RFC6298] recommends that the retransmission timer is restarted when an acknowledgment (ACK) that acknowledges new data is received and there is still outstanding data. The restart is conducted to guarantee that unacknowledged segments will be retransmitted after approximately RTO seconds. However, by restarting the timer on each incoming acknowledgment, retransmissions are not typically triggered RTO seconds after their previous transmission but rather RTO seconds after the last ACK arrived. The duration of this extra delay depends on several factors

but is in most cases approximately one RTT. Hence, in most situations the time before a retransmission is triggered is equal to "RTO + RTT".

The extra delay can be significant, especially for applications that use a lower RT_{min} than the standard of 1 second and/or in environments with high RTTs, e.g. mobile networks. The restart approach is illustrated in Figure 1 where a TCP sender transmits three segments to a receiver. The arrival of the first and second segment triggers a delayed ACK [RFC1122], which restarts the RTO timer at the sender. The RTO restart is performed approximately one RTT after the transmission of the third segment. Thus, if the third segment is lost, as indicated in Figure 1, the effective loss detection time is "RTO + RTT" seconds. In some situations, the effective loss detection time becomes even longer. Consider a scenario where only two segments are outstanding. If the second segment is lost, the time to expire the delayed ACK timer will also be included in the effective loss detection time.

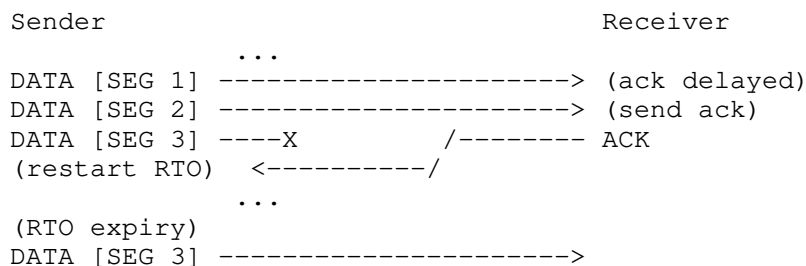


Figure 1: RTO restart example

During normal TCP bulk transfer the current RTO restart approach is not a problem. Actually, as long as enough segments arrive at a receiver to enable fast retransmit, RTO-based loss recovery should be avoided. RTOs should only be used as a last resort, as they drastically lower the congestion window compared to fast retransmit, and the current approach can therefore be beneficial -- it is described in [EL04] to act as a "safety margin" that compensates for some of the problems that the authors have identified with the standard RTO calculation. Notably, the authors of [EL04] also state that "this safety margin does not exist for highly interactive applications where often only a single packet is in flight."

There are only a few situations where timeouts are appropriate, or the only choice. For example, if the network is severely congested and no segments arrive, RTO-based recovery should be used. In this

situation, the time to recover from the loss(es) will not be the performance bottleneck. Furthermore, for connections that do not utilize enough capacity to enable fast retransmit, RTO is the only choice. The time needed for loss detection in such scenarios can become a serious performance bottleneck.

3. RTO Restart Algorithm

To enable faster loss recovery for connections that are unable to use fast retransmit, an alternative RTO restart can be used. By resetting the timer to "RTO - T_earliest", where T_earliest is the time elapsed since the earliest outstanding segment was transmitted, retransmissions will always occur after exactly RTO seconds. This approach makes the RTO more aggressive than the standardized approach in [RFC6298] but still conforms to the requirement in [RFC6298] that segments must not be retransmitted earlier than RTO seconds after their original transmission.

This document specifies the following update of step 5.3 in Section 5 of [RFC6298] (and a similar update in Section 6.3.2 of [RFC4960] for SCTP):

When an ACK is received that acknowledges new data:

- (1) Set T_earliest = 0.
- (2) If the following two conditions hold:
 - (a) The number of outstanding segments is less than four.
 - (b) There is no unsent data ready for transmission or the receiver's advertised window does not permit transmission.set T_earliest to the time elapsed since the earliest outstanding segment was sent.
- (3) Restart the retransmission timer so that it will expire after "RTO - T_earliest" seconds (for the current value of RTO).

The update requires TCP implementations to track the time elapsed since the transmission of the earliest outstanding segment (T_earliest). As the alternative restart is used only when the number of outstanding segments is less than four only four segments need to be tracked. Furthermore, some implementations of TCP (e.g. Linux TCP) already track the transmission times of all segments.

4. Discussion

The currently standardized algorithm has been shown to add at least one RTT to the loss recovery process in TCP [LS00] and SCTP [HB08][PBP09]. Applications that have strict timing requirements (e.g. telephony signaling and gaming) rather than throughput requirements may want to use a lower RTT_{min} than the standard of 1 second [RFC4166]. For such applications the modified restart approach could be important as the RTT and also the delayed ACK timer of receivers will be large components of the effective loss recovery time. Measurements in [HB08] have shown that the total transfer time of a lost segment (including the original transmission time and the loss recovery time) can be reduced with up to 35% using the suggested approach. These results match those presented in [PGH06][PBP09], where the modified restart approach is shown to significantly reduce retransmission latency.

There are several proposals that address the problem of not having enough ACKs for loss recovery. In what follows, we explain why the mechanism described here is complementary to these approaches:

The limited transmit mechanism [RFC3042] allows a TCP sender to transmit a previously unsent segment for each of the first two duplicate acknowledgments. By transmitting new segments, the sender attempts to generate additional duplicate acknowledgments to enable fast retransmit. However, limited transmit does not help if no previously unsent data is ready for transmission or if the receiver is out of buffer space. [RFC5827] specifies an early retransmit algorithm to enable fast loss recovery in such situations. By dynamically lowering the amount of duplicate acknowledgments needed for fast retransmit (dupthresh), based on the number of outstanding segments, a smaller number of duplicate acknowledgments are needed to trigger a retransmission. In some situations, however, the algorithm is of no use or might not work properly. First, if a single segment is outstanding, and lost, it is impossible to use early retransmit. Second, if ACKs are lost, the early retransmit cannot help. Third, if the network path reorders segments, the algorithm might cause more unnecessary retransmissions than fast retransmit.

TCP-NCR [RFC4653] sets the dupthresh to three or more, to better disambiguate reordered and lost segments. In addition, early retransmit lowers the dupthresh when the amount of outstanding data is small, to enable faster loss recovery. The reasons why the RTO restart procedure described in this document does not take dynamic dupthresh considerations into account are twofold. First, if a larger dupthresh is used, the RTO restart approach could be used when the congestion window, and the amount of outstanding data, is larger. However, in such situations the actual amount of outstanding data can

significantly impact the RTT of the connection, making it potentially dangerous to be more aggressive. Second, if a smaller dupthresh is used, the amount of outstanding data needed for a restart is smaller. However, as the congestion window is already small, it does not matter if a retransmission is due to a fast retransmit or an RTO. The resulting congestion window will still be very small, and the only difference is how quickly TCP infers segment loss.

Tail Loss Probe [TLP] is a proposal to send up to two "probe segments" when a timer fires which is set to a value smaller than the RTO. A "probe segment" is a new segment if new data is available, else a retransmission. The intention is to compensate for sluggish RTO behavior in situations where the RTO greatly exceeds the RTT, which, according to measurements reported in [TLP], is not uncommon. The Probe timeout (PTO) is at least 2 RTTs, and only scheduled in case the RTO is farther than the PTO. A spurious PTO is less risky than a spurious RTO, as it would not have the same negative effects (clearing the scoreboard and restarting with slow-start). In contrast, RTO restart is trying to make the RTO more appropriate in cases where there is no need to be overly cautious.

TLP could kick in in situations where RTO restart does not apply, and it could overrule (yielding a similar general behavior, but with a lower timeout) RTO restart in cases where the number of outstanding segments is smaller than 4 and no new segments are available for transmission. The shorter RTO from RTO restart also reduces the probability that TLP is activated because PTO might be farther than RTO. This could make RTO restart more aggressive than the algorithm in [TLP] when:

- (1) no data has been sent in an interval exceeding the RTO
- (2) the number of outstanding segments is 3
- (3) (defined in [RFC5681]) is at least 3

because, under these conditions, in accordance with [RFC5681], 3 packets can immediately be retransmitted, whereas TLP only allows up to two consecutive PTOs.

5. IANA Considerations

This memo includes no request to IANA.

6. Security Considerations

This document discusses a change in how to set the retransmission timer's value when restarted. This change does not raise any new security issues with TCP or SCTP.

7. References

7.1. Normative References

- [RFC1122] Braden, R., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, October 1989.
- [RFC2018] Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, "TCP Selective Acknowledgment Options", RFC 2018, October 1996.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC3042] Allman, M., Balakrishnan, H., and S. Floyd, "Enhancing TCP's Loss Recovery Using Limited Transmit", RFC 3042, January 2001.
- [RFC4166] Coene, L. and J. Pastor-Balbas, "Telephony Signalling Transport over Stream Control Transmission Protocol (SCTP) Applicability Statement", RFC 4166, February 2006.
- [RFC4653] Bhandarkar, S., Reddy, A., Allman, M., and E. Blanton, "Improving the Robustness of TCP to Non-Congestion Events", RFC 4653, August 2006.
- [RFC4960] Stewart, R., "Stream Control Transmission Protocol", RFC 4960, September 2007.
- [RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, September 2009.
- [RFC5827] Allman, M., Avrachenkov, K., Ayesta, U., Blanton, J., and P. Hurtig, "Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)", RFC 5827, May 2010.
- [RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, June 2011.

7.2. Informative References

- [BPS98] Balakrishnan, H., Padmanabhan, V., Seshan, S., Stemm, M., and R. Katz, "TCP Behavior of a Busy Web Server: Analysis and Improvements", Proc. IEEE INFOCOM Conf., March 1998.
- [EL04] Ekstroem, H. and R. Ludwig, "The Peak-Hopper: A New End-to-End Retransmission Timer for Reliable Unicast Transport", IEEE INFOCOM 2004, March 2004.
- [HB08] Hurtig, P. and A. Brunstrom, "SCTP: designed for timely message delivery?", Springer Telecommunication Systems, May 2010.
- [LS00] Ludwig, R. and K. Sklower, "The Eifel retransmission timer", ACM SIGCOMM Comput. Commun. Rev., 30(3), July 2000.
- [P09] Petlund, A., "Improving latency for interactive, thin-stream applications over reliable transport", Unipub PhD Thesis, Oct 2009.
- [PBP09] Petlund, A., Beskow, P., Pedersen, J., Paaby, E., Griwodz, C., and P. Halvorsen, "Improving SCTP Retransmission Delays for Time-Dependent Thin Streams", Springer Multimedia Tools and Applications, 45(1-3), 2009.
- [PGH06] Pedersen, J., Griwodz, C., and P. Halvorsen, "Considerations of SCTP Retransmission Delays for Thin Streams", IEEE LCN 2006, November 2006.
- [RJ10] Ramachandran, S., "Web metrics: Size and number of resources", Google <http://code.google.com/speed/articles/web-metrics.html>, May 2010.
- [TLP] Dukkipati, N., Cardwell, N., Cheng, Y., and M. Mathis, "TCP Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses", draft-dukkipati-tcpm-tcp-loss-probe-00.txt (work in progress), July 2012.

Authors' Addresses

Per Hurtig
Karlstad University
Universitetsgatan 2
Karlstad, 651 88
Sweden

Phone: +46 54 700 23 35
Email: per.hurtig@kau.se

Andreas Petlund
Simula Research Laboratory AS
P.O. Box 134
Lysaker, 1325
Norway

Phone: +47 67 82 82 00
Email: apetlund@simula.no

Michael Welzl
University of Oslo
PO Box 1080 Blindern
Oslo, N-0316
Norway

Phone: +47 22 85 24 20
Email: michawe@ifi.uio.no

TCP Maintenance (TCPM)
Internet-Draft
Obsoletes: 1323 (if approved)
Intended status: Standards Track
Expires: October 13, 2014

D. Borman
Quantum Corporation
B. Braden
University of Southern
California
V. Jacobson
Google, Inc.
R. Scheffenegger, Ed.
NetApp, Inc.
April 11, 2014

TCP Extensions for High Performance
draft-ietf-tcpm-1323bis-21

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 the TCP Window Scale (WS) option and the TCP Timestamps (TS) option and their semantics. The Window Scale option is used to support larger receive windows, while the Timestamps option can be used for at least two distinct mechanisms, PAWS (Protection Against Wrapped Sequences) and RTTM (Round Trip Time Measurement), that are also described herein.

This document obsoletes RFC1323 and describes changes from it.

Status of this Memo

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

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

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

This Internet-Draft will expire on October 13, 2014.

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	4
1.1. TCP Performance	4
1.2. TCP Reliability	5
1.3. Using TCP options	6
1.4. Terminology	7
2. TCP Window Scale option	8
2.1. Introduction	8
2.2. Window Scale option	8
2.3. Using the Window Scale option	9
2.4. Addressing Window Retraction	10
3. TCP Timestamps option	12
3.1. Introduction	12
3.2. Timestamps option	12
4. The RTTM Mechanism	15
4.1. Introduction	15
4.2. Updating the RTO value	16
4.3. Which Timestamp to Echo	16
5. PAWS - Protection Against Wrapped Sequence Numbers	20
5.1. Introduction	20
5.2. The PAWS Mechanism	20
5.3. Basic PAWS Algorithm	21
5.4. Timestamp Clock	23
5.5. Outdated Timestamps	25
5.6. Header Prediction	25
5.7. IP Fragmentation	27
5.8. Duplicates from Earlier Incarnations of Connection	27
6. Conclusions and Acknowledgments	28
7. Security Considerations	28
7.1. Privacy Considerations	30
8. IANA Considerations	30
9. References	30
9.1. Normative References	30
9.2. Informative References	31
Appendix A. Implementation Suggestions	34
Appendix B. Duplicates from Earlier Connection Incarnations	35
B.1. System Crash with Loss of State	35
B.2. Closing and Reopening a Connection	36
Appendix C. Summary of Notation	37
Appendix D. Event Processing Summary	38
Appendix E. Timestamps Edge Cases	43
Appendix F. Window Retraction Example	44
Appendix G. RTO calculation modification	45
Appendix H. Changes from RFC 1323	45
Authors' Addresses	48

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 have 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].

Changes between [RFC1323] and this document are detailed in Appendix H. These changes are partly due to errata in [RFC1323], and partly due to the improved understanding of how the involved components interact.

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. It is recommended that a modern TCP stack implements 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 two fundamental performance problems with basic 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 $2^{16} = 64$ KiB. For LFN paths where the bandwidth * delay product exceeds 64 KiB, the receive window limits the maximum throughput of the TCP connection over the path, i.e., the amount of unacknowledged data that TCP can send in order to keep the pipeline full.

To circumvent this problem, Section 2 of this memo defines a 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.

It must be noted, that the use of large receive windows increases the chance of too quickly wrapping sequence numbers, as described below in Section 1.2, (1).

(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 acknowledgments and their use are specified in separate documents, "TCP Selective Acknowledgment options" [RFC2018], "An Extension to the Selective Acknowledgement (SACK) option for TCP" [RFC2883], and "A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP" [RFC6675], and not further discussed in this document.

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 of large volumes of data (at least 4 GiB in the same session), 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 specification, as explained in Section 5.8 and Appendix B. In addition, the randomizing of ephemeral ports can also help to probabilistically reduce the chances of duplicates from earlier connections. However, case (1), avoiding the reuse of sequence numbers within the same connection, requires an upper bound on MSL that depends upon the transfer rate, and at high enough rates, a dedicated 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 3.2 to protect against old duplicates from the same connection.

1.3. Using TCP options

The extensions defined in this document all use TCP options.

When [RFC1323] was published, there was concern that some buggy TCP implementation might crash on the first appearance of an option on a non-<SYN> segment. However, bugs like that can lead to DOS attacks against a TCP. Research has shown that most TCP implementations will properly handle unknown options on non-<SYN> segments ([Medina04], [Medina05]). 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.

The window scale option negotiates fundamental parameters of the TCP session. Therefore, it is only sent during the initial handshake. Furthermore, the window scale 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 10 bytes (up to 12 bytes including padding) to the 20-byte TCP header. It is required that this TCP option will be sent on all non-<SYN> segments after an exchange of options on the <SYN> segments has

indicated that both sides understand this extension.

Research has shown that the use of the Timestamps option to take additional RTT samples within each RTT has little effect on the ultimate retransmission timeout value [Allman99]. However, there are other uses of the Timestamps option, such as the Eifel mechanism [RFC3522], [RFC4015], and PAWS (see Section 5) which improve overall TCP security and performance. The extra header bandwidth used by this option should be evaluated for the gains in performance and security in an actual deployment.

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 document are important for LFNs 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.

1.4. 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].

In this document, these words will appear with that interpretation only when in UPPER CASE. Lower case uses of these words are not to be interpreted as carrying [RFC2119] significance.

2. TCP Window Scale option

2.1. Introduction

The window scale extension expands the definition of the TCP window to 30 bits and then uses an implicit scale factor to carry this 30-bit value in the 16-bit Window field of the TCP header (SEG.WND in [RFC0793]). The exponent of the scale factor is carried in a 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.

2.2. 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 both send and receive window scaling, and (2) communicate the exponent of 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 and the exponent 0. The scale factor is limited to a power of two and encoded logarithmically, so it may be implemented by binary shift operations. The maximum scale exponent is limited to 14 for a maximum permissible receive window size of 1 GiB ($2^{(14+16)}$).

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). If a Window Scale option was received in the initial <SYN> segment, then this option MAY be sent in the <SYN,ACK> segment. A Window Scale option in a segment without a SYN bit MUST be ignored.

The window field in a segment where the SYN bit is set (i.e., a <SYN> or <SYN,ACK>) MUST NOT be scaled.

2.3. Using the Window Scale option

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

- o The connection state is augmented by two window shift counters, Snd.Wind.Shift and Rcv.Wind.Shift, 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 SHOULD send its own Window Scale option in the <SYN,ACK> segment.
- o The Window Scale option MUST be 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 MUST set Snd.Wind.Shift to S and MUST set Rcv.Wind.Shift to R; otherwise, it MUST set both Snd.Wind.Shift and Rcv.Wind.Shift to zero.
- o The window field (SEG.WND) in the header of every incoming segment, with the exception of <SYN> segments, MUST be left-shifted by Snd.Wind.Shift bits before updating SND.WND:

$$\text{SND.WND} = \text{SEG.WND} \ll \text{Snd.Wind.Shift}$$

(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, MUST be right-shifted by Rcv.Wind.Shift bits:

$$\text{SEG.WND} = \text{RCV.WND} \gg \text{Rcv.Wind.Shift}$$

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 maximum window size must be less than 2^{31} , or

$$\text{max window} < 2^{30}$$

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 $S \leq 14$. Thus, the shift count MUST be limited to 14 (which allows windows of $2^{30} = 1 \text{ GiB}$). If a Window Scale option is received with a shift.cnt value larger than 14, the TCP SHOULD log the error but MUST use 14 instead of the specified value. This is safe as a sender can always choose to only partially use any signaled receive window. If the receiver is scaling by a factor larger than 14 and the sender is only scaling by 14 then the receive window used by the sender will appear smaller than it is in reality.

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 (see [RFC5681]) is not affected by the scale factor, so window scaling will not introduce quantization into the congestion window.

2.4. Addressing Window Retraction

When a non-zero scale factor is in use, there are instances when a retracted window can be offered - see Appendix F for a detailed example. 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 be within the window announced by the most recent <ACK>.
- 4) On first retransmission, or if the sequence number is out-of-window by less than $2^{\text{Rcv.Wind.Shift}}$ then do normal retransmission(s) without regard to receiver window as long as the original segment was in window when it was sent.
- 5) Subsequent retransmissions MAY only be sent, if they are within the window announced by the most recent <ACK>.

3. TCP Timestamps option

3.1. Introduction

The Timestamps option is introduced to address some of the issues mentioned in Section 1.1 and Section 1.2. The Timestamps option is 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. Originally used primarily for timestamping individual segments, the properties of the Timestamps option allow not only the use for taking time measurements (Section 4), but additional uses as well (Section 5).

It is necessary to remember that there is a distinction between the Timestamps option conveying timestamp information, and the use of that information. In particular, the Round Trip Time Measurement (RTTM) mechanism must be viewed independently from updating the Retransmission Timeout (RTO) (see Section 4.2). In this case, the sample granularity also needs to be taken into account. Other mechanisms, such as PAWS, or Eifel, are not built upon the timestamp information itself, but are based on the intrinsic property of monotonically non-decreasing values.

The Timestamps option is important when large receive windows are used, to allow the use of the PAWS mechanism (see Section 5). Furthermore, the option may be useful for all TCPs, since it simplifies the sender and allows the use of additional optimizations such as Eifel ([RFC3522], [RFC4015]) and others ([RFC6817], [Kuzmanovic03], [Kuehlewind10]).

3.2. 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 (TSecr) field is valid if the ACK bit is set in the TCP header. If the ACK bit is not set in the outgoing TCP header, the sender of that segment SHOULD set the TSecr field to zero. When the ACK bit is set in an outgoing segment, the sender MUST echo a recently received Timestamp Value (TSval) sent by the remote TCP in the TSval field of a Timestamps option. The exact rules on which TSval MUST be echoed are given in Section 4.3. When the ACK bit is not set, the receiver MUST ignore the value of the TSecr field.

A TCP MAY send the Timestamps option (TSopt) in an initial <SYN> segment (i.e., segment containing a SYN bit and no ACK bit), and MAY send a TSopt in <SYN,ACK> only if it received a TSopt in the initial <SYN> segment for the connection.

Once TSopt has been successfully negotiated, that is both <SYN>, and <SYN,ACK> contain TSopt, the TSopt MUST be sent in every non-<RST> segment for the duration of the connection, and SHOULD be sent in an <RST> segment (see Section 5.2 for details). The TCP SHOULD remember this state by setting a flag, referred to as Snd.TS.OK, to one. If a non-<RST> segment is received without a TSopt, a TCP SHOULD silently drop the segment. A TCP MUST NOT abort a TCP connection because any segment lacks an expected TSopt.

Implementations are strongly encouraged to follow the above rules for handling a missing Timestamps option, and the order of precedence mentioned in Section 5.3 when deciding on the acceptance of a segment.

If a receiver chooses to accept a segment without an expected Timestamps option, it must be clear that undetectable data corruption may occur.

Such a TCP receiver may experience undetectable wrapped- sequence effects, such as data (payload) corruption or session stalls. In order to maintain the integrity of the payload data, in particular on high speed networks, it is paramount to follow the described processing rules.

However, it has been mentioned that under some circumstances, the above guidelines are too strict, and some paths sporadically suppress the Timestamps option, while maintaining payload integrity. A path behaving in this manner should be deemed unacceptable, but it has been noted that some implementations relax the acceptance rules as a workaround, and allow TCP to run across such paths [Oppermann13]

If a TSopt is received on a connection where TSopt was not negotiated in the initial three-way handshake, the TSopt MUST be ignored and the packet processed normally.

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

TSopt is required for the two mechanisms described in sections 4 and 5. There are also other mechanisms that rely on the presence of the TSopt, e.g. [RFC3522]. If a TCP stopped sending TSopt at any time during an established session, it interferes with these mechanisms. This update to [RFC1323] describes explicitly the previous assumption (see Section 5.2), that each TCP segment must have TSopt, once negotiated.

4. The RTTM Mechanism

4.1. Introduction

One use of the Timestamps option is to measure the round trip time of virtually every packet acknowledged. The Round Trip Time Measurement (RTTM) mechanism requires a Timestamps option in every measured 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.

TCP measures the round trip time (RTT), primarily for the purpose of arriving at a reasonable value for the Retransmission Timeout (RTO) timer interval. Accurate and current RTT estimates are necessary to adapt to changing traffic conditions, while a conservative estimate of the RTO interval is necessary to minimize spurious RTOs.

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 an 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 MAY be used to update the averaged RTT measurement only if the segment advances the left edge of the send window, i.e. SND.UNA is increased.

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.

4.2. Updating the RTO value

When [RFC1323] was originally written, it was perceived that taking RTT measurements for each segment, and also during retransmissions, would contribute to reduce spurious RTOs, while maintaining the timeliness of necessary RTOs. At the time, RTO was also the only mechanism to make use of the measured RTT. It has been shown, that taking more RTT samples has only a very limited effect to optimize RTOs [Allman99].

Implementers should note that with timestamps multiple RTTMs can be taken per RTT. The [RFC6298] RTO estimator has weighting factors, alpha and beta, based on an implicit assumption that at most one RTTM will be sampled per RTT. When multiple RTTMs per RTT are available to update the RTO estimator, an implementation SHOULD try to adhere to the spirit of the history specified in [RFC6298]. An implementation suggestion is detailed in Appendix G.

[Ludwig00] and [Floyd05] have highlighted the problem that an unmodified RTO calculation, which is updated with per-packet RTT samples, will truncate the path history too soon. This can lead to an increase in spurious retransmissions, when the path properties vary in the order of a few RTTs, but a high number of RTT samples are taken on a much shorter timescale.

4.3. 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 TCPs acknowledge only every second 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 <ACK>s 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 RCV.NXT.

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 and advances the window represents the most recent measurement of the network characteristics. An 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 <ACK>s 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 Segments arrive in sequence, and some of the <ACK>s 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 Segments arrive out of order, and every segment 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

Another use for the Timestamps options is the mechanism to Protect Against Wrapped Sequence numbers (PAWS). Section 5.2 describes a simple mechanism to reject old duplicate segments that might corrupt an open TCP connection. PAWS operates within a single TCP connection, using state that is saved in the connection control block. Section 5.8 and Appendix H 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 TCP Timestamps option 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 the PAWS 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 non-decreasing. For example, an implementation 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. For simplicity, the value TS.Recent introduced in Section 4.3 is used instead, as using a common value for both PAWS and RTTM simplifies the implementation. As Section 4.3 explained, TS.Recent differs from the timestamp from the last in-sequence segment only in the case of delayed <ACK>s, and therefore by less than one window. Either choice will therefore protect against sequence number wrap-around.

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 <ACK>s: 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> segments.

While still under discussion, to enable research into this area it is now RECOMMENDED that when generating an <RST>, that if the segment 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 segment and SEG.TSval SHOULD be set to zero. If an <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 an <RST> segment is received, it MUST NOT be subjected to the PAWS check by verifying an acceptable value in SEG.TSval, 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.

5.3. Basic PAWS Algorithm

If the PAWS algorithm is used, the following processing MUST be performed on all incoming segments for a synchronized connection. Also, PAWS processing MUST take precedence over the regular TCP acceptability check (Section 3.3 in [RFC0793]), which is performed after verification of the received Timestamps option:

- 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 acknowledgment 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 4.3), 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 **MUST** be 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 segment will show up out-of-sequence to be queued at the receiver (e.g., up to 2^{30} bytes of data); the Timestamps 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 <ACK>s, then the queued segments that caused these <ACK>s must have been received already.

Even if a segment were delayed past the RTO, the Fast Retransmit mechanism [Jacobson90c] will cause the delayed segments 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.

5.4. 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 as a source of monotonic non-decreasing values to stamp the segments. The receiver treats the timestamp value as simply a monotonically non-decreasing serial number, without any connection to time. 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 2.2, 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 5.5), 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.

- o Some hosts have a hardware clock that is guaranteed to be monotonic between hardware resets.
- o A clock interrupt may be used to simply increment a binary integer by 1 periodically.
- o 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).

- o A random offset may be added to the timestamp clock on a per connection basis. See [RFC6528], section 3, on randomizing the initial sequence number (ISN). The same function with a different secret key can be used to generate the per connection timestamp offset.

5.5. 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.

5.6. 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 100 Mbit/s 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.

5.7. IP Fragmentation

At high data rates, the protection against old segments 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 segments 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.

5.8. 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 Acknowledgments

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 stacks that do not implement the extensions.

These mechanisms are implemented using 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 segments. 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 segments 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.

See [RFC5961] for mitigation strategies to blind in-window attacks.

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. It is therefore RECOMMENDED to generate a random, per-connection offset to be used with the clock source when generating the Timestamps option value (see Section 5.4). By carefully choosing this random offset, further improvements as described in [RFC6191] are possible.

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

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

Middleboxes and TCP options:

Some middleboxes have been known to remove the TCP options described in this document from TCP segments [Hondall]. Middleboxes that remove TCP options described in this document from the <SYN> segment interfere with the selection of parameters appropriate for the session. Removing any of these options in a <SYN,ACK> segment will leave the end hosts in a state that destroys the proper operation of the protocol.

- * If a Window Scale option is removed from a <SYN,ACK> segment, the end hosts will not negotiate the window scaling factor correctly. Middleboxes must not remove or modify the Window Scale option from <SYN,ACK> segments.
- * If a stateful firewall uses the window field to detect whether a received segment is inside the current window, and does not support the Window Scale option, it will not be able to correctly determine whether or not a packet is in the window. These middle boxes must also support the Window Scale option and apply the scale factor when processing segments. If the window scale factor cannot be determined, it must not do window based processing.
- * If the Timestamps option is removed from the <SYN> or <SYN,ACK> segment, high speed connections that need PAWS would not have that protection. Successful negotiation of Timestamps option enforces a stricter verification of incoming segments at the receiver. If the Timestamps option was removed from a subsequent data segment after a successful negotiation (e.g. as part of re-segmentation), the segment is discarded by the receiver without further processing. Middleboxes should not remove the Timestamps option.

- * It must be noted that [RFC1323] doesn't address the case of the Timestamps option being dropped or selectively omitted after being negotiated, and that the update in this document may cause some broken middlebox behavior to be detected (potentially unresponsive TCP sessions).

Implementations that depend on PAWS 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. This is not just to protect the connection against middleboxes that might remove the Timestamps option, but also against remote hosts that do not have Timestamp support.

7.1. Privacy Considerations

The TCP options described in this document do not expose individual users data. However, a naive implementation simply using the system clock as source for the Timestamps option will reveal characteristics of the TCP potentially allowing more targeted attacks. It is therefore RECOMMENDED to generate a random, per-connection offset to be used with the clock source when generating the Timestamps option value (see Section 5.4).

Furthermore, the combination, relative ordering and padding of the TCP options described in Section 2.2 and Section 3.2 will reveal additional clues to allow the fingerprinting of the system.

8. IANA Considerations

The described TCP options are well known from the superceded [RFC1323]. IANA is requested to update the "TCP Option Kind Numbers" table under "TCP parameters" to list <this-RFC-to-be> as the reference for the options "WSopt - Window Scale Option" and "TSopt - Timestamps Option".

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

- [Allman99] Allman, M. and V. Paxson, "On Estimating End-to-End Network Path Properties", Proc. ACM SIGCOMM Technical Symposium, Cambridge, MA, September 1999, <<http://aciri.org/mallman/papers/estimation-1a.pdf>>.
- [Floyd05] Floyd, S., "[tcpm] How the RTO should be estimated with timestamps", Message from 26.Jan.2007 to the tcpm mailing list, August 2005, <<http://www.ietf.org/mail-archive/web/tcpm/current/msg02508.html>>.
- [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>>.
- [Honda11] Honda, M., Nishida, Y., Raiciu, C., Greenhalgh, A., Handley, M., and H. Tokuda, "Is it still possible to extend TCP?", Proc. of ACM Internet Measurement Conference (IMC) '11, November 2011.
- [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>>.
- [Karn87] Karn, P. and C. Partridge, "Estimating Round-Trip Times in Reliable Transport Protocols", Proc. SIGCOMM '87, August 1987.

- [Kuehlewind10]
Kuehlewind, M. and B. Briscoe, "Chirping for Congestion Control - Implementation Feasibility", November 2010, <bobbriscoe.net/projects/netsvc_i-f/chirp_pfldnet10.pdf>.
- [Kuzmanovic03]
Kuzmanovic, A. and E. Knightly, "TCP-LP: Low-Priority Service via End-Point Congestion Control", 2003, <www.cs.northwestern.edu/~akuzma/doc/TCP-LP-ToN.pdf>.
- [Ludwig00]
Ludwig, R. and K. Sklower, "The Eifel Retransmission Timer", ACM SIGCOMM Computer Communication Review Volume 30 Issue 3, July 2000, <<http://ccr.sigcomm.org/archive/2000/july00/LudwigFinal.pdf>>.
- [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>>.
- [Medina04]
Medina, A., Allman, M., and S. Floyd, "Measuring Interactions Between Transport Protocols and Middleboxes", Proc. ACM SIGCOMM/USENIX Internet Measurement Conference, October 2004, August 2004, <<http://www.icir.net/tbit/tbit-Aug2004.pdf>>.
- [Medina05]
Medina, A., Allman, M., and S. Floyd, "Measuring the Evolution of Transport Protocols in the Internet", ACM Computer Communication Review 35(2), April 2005, <<http://icir.net/floyd/papers/TCPEvolution-Mar2005.pdf>>.
- [Oppermann13]
Oppermann, A., "[tcpm] Explanation to the relaxation of TSopt acceptance rules", Message to the tcpm mailing list, Jun 2013, <<http://www.ietf.org/mail-archive/web/tcpm/current/msg08001.html>>.
- [RFC1072] Jacobson, V. and R. Braden, "TCP extensions for long-delay paths", RFC 1072, October 1988.
- [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.
- [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.
- [RFC3522] Ludwig, R. and M. Meyer, "The Eifel Detection Algorithm for TCP", RFC 3522, April 2003.
- [RFC4015] Ludwig, R. and A. Gurtov, "The Eifel Response Algorithm for TCP", RFC 4015, February 2005.
- [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.
- [RFC5961] Ramaiah, A., Stewart, R., and M. Dalal, "Improving TCP's Robustness to Blind In-Window Attacks", RFC 5961, August 2010.
- [RFC6191] Gont, F., "Reducing the TIME-WAIT State Using TCP Timestamps", BCP 159, RFC 6191, April 2011.
- [RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, June 2011.
- [RFC6528] Gont, F. and S. Bellovin, "Defending against Sequence Number Attacks", RFC 6528, February 2012.
- [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.

[RFC6817] Shalunov, S., Hazel, G., Iyengar, J., and M. Kuehlewind, "Low Extra Delay Background Transport (LEDBAT)", RFC 6817, December 2012.

Appendix A. Implementation Suggestions

TCP Option Layout

The following layout is 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 (64 KiB - 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 segment, then the user will remain in urgent mode until the next TCP segment arrives. That segment 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, then 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 Timestamps 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 $2 \times \text{MSL}$ 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 RTT 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.Shift: Receive window scale exponent
Snd.Wind.Shift: Send window scale exponent
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. 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.Shift>

...

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.Shift>. 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.Shift).

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.Shift and set Snd.WS.OK flag on. Otherwise, set both Snd.Wind.Shift and Rcv.Wind.Shift 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.Shift> 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/compartments and precedence are acceptable then, RCV.NXT is set to SEG.SEQ+1, IRS is set to SEG.SEQ, and any acknowledgments 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.Shift; otherwise, set both Snd.Wind.Shift and Rcv.Wind.Shift 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.Shift> 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.Shift,

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=SEND.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 $\text{SND.UNA} < \text{SEG.ACK} \leq \text{SND.NXT}$ then, set $\text{SND.UNA} \leftarrow \text{SEG.ACK}$. Also compute a new estimate of round-trip time. If Snd.TS.OK bit is on, use $\text{Snd.TSclock} - \text{SEG.TSecr}$; 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:

$\langle \text{SEQ} = \text{SND.NXT} \rangle \langle \text{ACK} = \text{RCV.NXT} \rangle \langle \text{CTL} = \text{ACK} \rangle$

If the Snd.TS.OK bit is on, include Timestamps option $\langle \text{TSval} = \text{Snd.TSclock}, \text{TSecr} = \text{TS.Recent} \rangle$ in this $\langle \text{ACK} \rangle$ 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 E. 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 segments. 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 $\langle \text{ACK} \rangle$ is lost, and before the retransmission timer fires, another returning $\langle \text{ACK} \rangle$

segment arrives, which acknowledges 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 segment losses, and the connection will have other more serious operational problems than using an inflated RTTM in the RTO calculation.

Appendix F. Window Retraction Example

Consider an established TCP connection using a scale factor of 128, Snd.Wind.Shift=7 and Rcv.Wind.Shift=7, that is running with a very small window because the receiver is bottlenecked and both ends are doing small reads and writes.

Consider the ACKs coming back:

SEG.ACK	SEG.WIN	computed	SND.WIN	receiver's actual window
1000	2	1256		1300

The sender writes 40 bytes and receiver ACKs:

1040	2	1296		1300
------	---	------	--	------

The sender writes 5 additional bytes and the receiver has a problem. Two choices:

1045	2	1301	1300	- BEYOND BUFFER
1045	1	1173	1300	- RETRACTED WINDOW

This is a general problem and can happen any time the sender does a write which is smaller than the window scale factor.

In most stacks it is at least partially obscured when the window size is larger than some small number of segments because the stacks prefer to announce windows that are an integral number of segments, rounded up to the next scale factor. This plus silly window suppression tends to cause less frequent, larger window updates. If the window was rounded down to a segment size there is more opportunity to advance the window, the BEYOND BUFFER case above, rather than retracting it.

Appendix G. RTO calculation modification

Taking multiple RTT samples per window would shorten the history calculated by the RTO mechanism in [RFC6298], and the below algorithm aims to maintain a similar history as originally intended by [RFC6298].

It is roughly known how many samples a congestion window worth of data will yield, not accounting for ACK compression, and ACK losses. Such events will result in more history of the path being reflected in the final value for RTO, and are uncritical. This modification will ensure that a similar amount of time is taken into account for the RTO estimation, regardless of how many samples are taken per window:

```
ExpectedSamples = ceiling(FlightSize / (SMSS * 2))
```

```
alpha' = alpha / ExpectedSamples
```

```
beta' = beta / ExpectedSamples
```

Note that the factor 2 in ExpectedSamples is due to "Delayed ACKs".

Instead of using alpha and beta in the algorithm of [RFC6298], use alpha' and beta' instead:

```
RTTVAR <- (1 - beta') * RTTVAR + beta' * |SRTT - R'|
```

```
SRTT <- (1 - alpha') * SRTT + alpha' * R'
```

```
(for each sample R')
```

Appendix H. Changes from RFC 1323

Several important updates and clarifications to the specification in RFC 1323 are made in these document. The technical changes are summarized below:

- (a) A wrong reference to SND.WND was corrected to SEG.WND in Section 2.3
- (b) Section 2.4 was added describing the unavoidable window retraction issue, and explicitly describing the mitigation steps necessary.
- (c) In Section 3.2 the wording how the Timestamps option negotiation is to be performed was updated with RFC2119 wording. Further, a number of paragraphs were added to clarify the expected behavior with a compliant implementation using TSopt, as RFC1323 left room for interpretation - e.g. potential late enablement of TSopt.
- (d) 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 segment, or the retransmitted segment. With timestamps, that ambiguity is removed since the TSecr in the <ACK> will contain the TSval from whichever data segment made it to the destination.
- (e) RTTM update processing explicitly excludes segments not updating SND.UNA. The original text could be interpreted to allow taking RTT samples when SACK acknowledges some new, non-continuous data.
- (f) 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.
- (g) It is now recommended that the Timestamps option is included in <RST> segments if the incoming segment contained a Timestamps option.
- (h) <RST> segments are explicitly excluded from PAWS processing.

- (i) Added text to clarify the precedence between regular TCP [RFC0793] and this document Timestamps option / PAWS processing. Discussion about combined acceptability checks are ongoing.
- (j) 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 timestamp values to be randomized on a per-connection basis. Setting Snd.TSoffset to zero yields the same results as [RFC1323]. Text was added to guide implementers to the proper selection of these offsets, as entirely random offsets for each new connection will conflict with PAWS.
- (k) 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].
- (l) One correction was made to the Event Processing Summary in Appendix D. In SEND CALL/ESTABLISHED STATE, RCV.WND is used to fill in the SEG.WND value, not SND.WND.
- (m) Appendix G was added to exemplify how an RTO calculation might be updated to properly take the much higher RTT sampling frequency enabled by the Timestamps option into account.

Editorial changes of the document, that don't impact the implementation or function of the mechanisms described in this document include:

- (a) Removed much of the discussion in Section 1 to streamline the document. However, detailed examples and discussions in Section 2, Section 3 and Section 5 are kept as guideline for implementers.
- (b) Added short text that the use of WS increases the chances of sequence number wrap, thus the PAWS mechanism is required in certain environments.
- (c) Removed references to "new" options, as the options were introduced in [RFC1323] already. Changed the text in Section 1.3 to specifically address TS and WS options.
- (d) Section 1.4 was added for [RFC2119] wording. Normative text was updated with the appropriate phrases.

- (e) Added < > brackets to mark specific types of segments, and replaced most occurrences of "packet" with "segment", where TCP segments are referred to.
- (f) Updated the text in Section 3 to take into account what has been learned since [RFC1323].
- (g) Removed some unused references.
- (h) Removed the list of changes between [RFC1323] and prior versions. These changes are mentioned in Appendix C of [RFC1323].
- (i) Moved Appendix Changes from RFC 1323 to the end of the appendices for easier lookup. In addition, the entries were split into a technical and an editorial part, and sorted to roughly correspond with the sections in the text where they apply.

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
Google, Inc.
1600 Amphitheatre Parkway
Mountain View CA 94043
USA

Email: vanj@google.com

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

Email: rs@netapp.com

Internet Draft
draft-ietf-tcpm-fastopen-10.txt
Intended status: Experimental
Expiration date: April, 2015

Y. Cheng
J. Chu
S. Radhakrishnan
A. Jain
Google
September 29, 2014

TCP Fast Open

Abstract

This document describes an experimental TCP mechanism TCP Fast Open (TFO). TFO allows data to be carried in the SYN and SYN-ACK packets and consumed by the receiving end during the initial connection handshake, and saves up to one full round trip time (RTT) compared to the standard TCP, which requires a three-way handshake (3WHS) to complete before data can be exchanged. However TFO deviates from the standard TCP semantics since the data in the SYN could be replayed to an application in some rare circumstances. Applications should not use TFO unless they can tolerate this issue detailed in the Applicability section.

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/lid-abstracts.html>

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

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Terminology	3
2. Data In SYN	3
2.1 Relaxing TCP Semantics on Duplicated SYNs	4
2.2. SYNs with Spoofed IP Addresses	4
3. Protocol Overview	5
4. Protocol Details	6
4.1. Fast Open Cookie	6
4.1.1. Fast Open option	7
4.1.2. Server Cookie Handling	7
4.1.3. Client Cookie Handling	8
4.1.3.1 Client Caching Negative Responses	9
4.2. Fast Open Protocol	9
4.2.1. Fast Open Cookie Request	10
4.2.2. TCP Fast Open	11
5. Security Considerations	13
5.1. Resource Exhaustion Attack by SYN Flood with Valid Cookies	13
5.1.1 Attacks from behind Shared Public IPs (NATs)	14
5.2. Amplified Reflection Attack to Random Host	14
6. TFO's Applicability	15
6.1 Duplicate Data in SYNs	15
6.2 Potential Performance Improvement	16
6.3. Example: Web Clients and Servers	16
6.3.1. HTTP Request Replay	16
6.3.2. HTTP over TLS (HTTPS)	16
6.3.3. Comparison with HTTP Persistent Connections	17
6.3.4. Load Balancers and Server farms	17
7. Open Areas for Experimentation	17
7.1. Performance impact due to middle-boxes and NAT	18
7.2. Impact on congestion control	18
7.3. Cookie-less Fast Open	18
8. Related Work	19
8.1. T/TCP	19
8.2. Common Defenses Against SYN Flood Attacks	19
8.3. Speculative Connections by the Applications	19

8.4. Fast Open Cookie in FIN	19
8.5. TCP Cookie Transaction (TCPCT)	20
9. IANA Considerations	20
10. Acknowledgement	20
11. References	20
11.1. Normative References	20
11.2. Informative References	21
Appendix A. Example Socket API Changes to support TFO	22
A.1 Active Open	22
A.2 Passive Open	23
Authors' Addresses	23

1. Introduction

TCP Fast Open (TFO) is an experimental update to TCP that 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, potential new security vulnerabilities and their mitigation, and the new socket API.

TFO is motivated by the performance needs of today's Web applications. Current TCP only permits data exchange after the 3-way handshake (3WHS) [RFC793], which adds one RTT to network latency. For short Web transfers this additional RTT is a significant portion of overall network latency, even when HTTP persistent connection is widely used. For example, the Chrome browser [Chrome] keeps TCP connections idle for up to 5 minutes but 35% of HTTP requests are made on new TCP connections [RCCJR11]. For such Web and Web-like applications placing data in the SYN can yield significant latency improvements. Next we describe how we resolve the challenges that arise upon doing so.

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 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.

2. Data In SYN

Standard TCP already allows data to be carried in SYN packets ([RFC793], section 3.4) but forbids the receiver from delivering it to the application until 3WSH is completed. This is because TCP's initial handshake serves to capture old or duplicate SYNs.

To enable applications exchange data in TCP handshake, TFO removes the constraint and allows data in SYN packets to be delivered to the application. This change of TCP semantic raises two issues discussed in the following subsections, making TFO unsuitable for certain applications.

Therefore TCP implementations MUST NOT use TFO by default, but only use TFO if requested explicitly by the application on a per service port basis. Applications need to evaluate TFO applicability described in Section 6 before using TFO.

2.1 Relaxing TCP Semantics on Duplicated SYNs

TFO allows data to be delivered to the application before the 3WSH is completed, thus opening itself to a data integrity issue in either of the two cases below:

- a) the receiver host receives data in a duplicate SYN after it has forgotten it received the original SYN (e.g. due to a reboot);
- b) the duplicate is received after the connection created by the original SYN has been closed and the close was initiated by the sender (so the receiver will not be protected by the 2MSL TIMEWAIT state).

The now obsoleted T/TCP [RFC1644] attempted to address these issues. It was not successful and not deployed due to various vulnerabilities as described in the Related Work section. 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 use to a class of applications (Section 6) that are tolerant of duplicate SYN packets with data. We believe this is the right design trade-off balancing complexity with usefulness.

2.2. SYNs with Spoofed IP Addresses

Standard TCP suffers from the SYN flood attack [RFC4987] because 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.

TFO goes one step further to allow server-side TCP to send up data to

the application layer before 3WSH is completed. This opens up 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 further mount an amplified reflection attack against victims of choice beyond the TFO server itself.

Numerous mitigation techniques against regular SYN flood attacks exist and have been well documented [RFC4987]. Unfortunately none are applicable to TFO. We propose a server-supplied cookie to mitigate these new vulnerabilities in Section 3 and evaluate the effectiveness of the defense in Section 7.

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 3WSH:

Requesting a Fast Open Cookie:

1. The client sends a SYN with a Fast Open option with an empty cookie field to request a cookie.
2. The server generates a cookie and sends it through the Fast Open 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 data and the cookie in the Fast Open option.
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 maximum 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 (caches cookie C)	<----- <SYN, ACK, CookieOpt=C> -----	SYN-RCVD

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

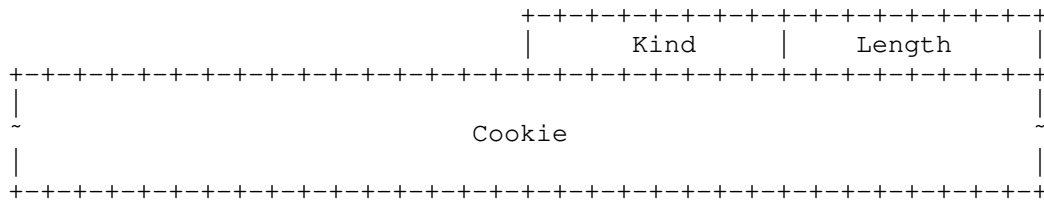
4. Protocol Details

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. Fast Open option



Kind 1 byte: constant-TBD (to be 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 0, or 4 to 16 bytes (Length - 2)

The Fast Open option is used to request or to send a Fast Open Cookie. When cookie is not present or empty, the option is used by the client to request a cookie from the server. When the cookie is present, the option is used to pass the cookie from the server to the client or from the client back to the server (to perform a Fast Open).

The minimum Cookie size is 4 bytes. Although the diagram shows a cookie aligned on 32-bit boundaries, alignment is not required. Options with invalid Length values or without SYN 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. We use SHOULD because in some cases the cookie may be trivially generated as discussed in Section 7.3.

1. The cookie authenticates the client's (source) IP address of the SYN packet. The IP address may 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 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 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 [RCCJR11].

If only one valid cookie is allowed per-IP 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.

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 dependent on the client and server IP addresses. Hence the client should cache at most one (most recently received) cookie per client and server IP addresses pair.

When caching cookies, we recommend that the client also cache the Maximum Segment Size (MSS) advertised by the server. The client can cache the MSS advertised by the server in order to determine the maximum amount of data that the client can fit in the SYN packet in

subsequent TFO connections. Caching the server MSS is useful because with Fast Open a client sends data in the SYN packet before the server announces its MSS in the SYN-ACK packet. If the client sends more data in the SYN packet than the server will accept, this will likely require the client to retransmit some or all of the data. Hence caching the server MSS can enhance performance.

Without a cached server MSS, the amount of data in the SYN packet is limited to the default MSS of 536 bytes for IPv4 [RFC1122] and 1240 bytes for IPv6 [RFC2460]. Even if the client complies with this limit when sending the SYN, it is known that an IPv4 receiver advertising an MSS less than 536 bytes can receive a segment larger than it is expecting.

If the cached MSS is larger than the typical size (1460 bytes for IPv4, or 1440 bytes for IPv6), then the excess data in the SYN packet may cause problems that offset the performance benefit of Fast Open. For example, the unusually large SYN may trigger IP fragmentation and may confuse firewalls or middleboxes, causing SYN retransmission and other side effects. Therefore the client MAY limit the cached MSS to 1460 bytes for IPv4 or 1440 for IPv6.

4.1.3.1 Client Caching Negative Responses

The client MUST cache negative responses from the server in order to avoid potential connection failures. Negative responses include server not acknowledging the data in SYN, ICMP error messages, and most importantly no response (SYN/ACK) from the server at all, i.e., connection timeout. The last case is likely due to incompatible middle-boxes or firewall blocking the connection completely after it sees data in SYN. If the client does not react to these negative responses and continue to retry Fast Open, the client may never be able to connect to the specific server.

For any negative responses, the client SHOULD disable Fast Open on the specific path (the source and destination IP addresses and ports) at least temporarily. Since TFO is enabled on a per-service port basis but cookies are independent of service ports, the client's 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 socket (IP address & 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 MUST 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 connection 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 option with a length field of 0 (empty cookie field).
2. The server responds with a SYN-ACK based on the procedures in the "Server Cookie Handling" section. This SYN-ACK may contain a Fast Open option if the server currently supports TFO for this listener port.
3. If the SYN-ACK has a Fast Open option with a cookie, 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, 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 will cause SYN or SYN-ACK timeouts. We RECOMMEND both the client and the server to 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 penalty (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 3WSH is justified if the SYN drop is due to network congestion). The next section describes a heuristic to detect such drops when the client receives the SYN-ACK.

We also RECOMMEND the client to record the set of servers that failed to respond to cookie requests and only attempt another cookie request after certain period.

4.2.2. TCP Fast Open

Once the client obtains the cookie from the target server, it can perform subsequent TFO connections until the cookie is expired by the server.

Client: Sending SYN

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

1. Send a SYN packet.
 - a. If the SYN packet does not have enough option space for the Fast Open option, abort TFO and fall back to regular 3WSHs.
 - b. Otherwise, include the Fast Open 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.

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 options.

Server: Receiving SYN and responding with SYN-ACK

Upon receiving the SYN packet with Fast Open 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 up to the implementation.

6. Advance to the SYN-RCVD state. If the FastOpened flag is set, the server MUST follow [RFC5681] (based on [RFC3390]) to set the initial congestion window for sending more data packets.

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

A special case is simultaneous open where the SYN receiver is a client in SYN-SENT state. The protocol remains the same because [RFC793] already supports both data in SYN and simultaneous open. But the client's socket may have data available to read before it's connected. This document does not cover the corresponding API change.

Client: Receiving SYN-ACK

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

1. If the SYN-ACK has a Fast Open option or MSS option or both, update the corresponding cookie and MSS information in the cookie cache.
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. 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 although generating bogus cookies is cost-free, the cost of validating the cookies, inherent to any authentication scheme, may be substantial compared to processing a regular SYN packet. We describe these new vulnerabilities of TFO and the countermeasures in detail below.

5.1. Resource Exhaustion Attack by SYN Flood with Valid Cookies

An 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). 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. Such an attack will show up on application server logs as an application level DoS from Bot-nets, triggering other defenses and alerts.

To protect the server it is important to limit the maximum number of total pending TFO connection requests, i.e., PendingFastOpenRequests (Section 4.2). 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.

The main impact of SYN floods against the standard TCP stack is not directly from the floods 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 floods 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 floods 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 queue length. 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.

5.1.1 Attacks from behind Shared Public IPs (NATs)

An attacker behind a NAT can easily obtain valid cookies to launch the above attack to hurt other clients that share the path. [BRISCOE12] 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 [RFC1323] to send non-zero Timestamp Echo Reply in SYN, potentially causing firewall issues. Therefore we believe the benefit does not outweigh the drawbacks.

5.2. 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. But in some cases it may be relatively easy.

The attacker here has little interest in mounting an attack on the victim host that has already been compromised. But it may be motivated to disrupt the victim's network. Since a stolen cookie is only valid for a single server, it 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.

One can argue that if the attacker has compromised the target network or hosts, it 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.

For example with DHCP an attacker can obtain cookies when he (or the host he has compromised) owns a particular IP address by performing regular Fast Open to servers supporting TFO and collect valid cookies. The attacker then actively or passively releases his IP address. When the IP address is re-assigned to a victim, the attacker now owning a different IP address, floods spoofed Fast Open requests to perform an amplified reflection attack on the victim.

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. TFO's Applicability

This section is to help applications considering TFO to evaluate TFO's benefits and drawbacks using the Web client and server applications as an example throughout. Applications here refer specifically to the process that writes data into the socket. For example a JavaScript process that sends data to the server. A proposed socket API change is in the Appendix.

6.1 Duplicate Data in SYNs

It is possible that using TFO results in the first data written to a socket to be delivered more than once to the application on the remote host (Section 2.1). This replay potential only applies to data in the SYN but not subsequent data exchanges.

Empirically [JIDKT07] showed the packet duplication on a Tier-1

network is rare. Since the replay only happens specifically when the SYN data packet is duplicated and also the duplicate arrives after the receiver has cleared the original SYN's connection state, the replay is thought to be uncommon in practice. Nevertheless a client that cannot handle receiving the same SYN data more than once MUST NOT enable TFO to send data in a SYN. Similarly a server that cannot accept receiving the same SYN data more than once MUST NOT enable TFO to receive data in a SYN. Further investigation is needed to judge about the probability of receiving duplicated SYN or SYN-ACK with data in non-Tier 1 networks.

6.2 Potential Performance Improvement

TFO is designed for latency-conscious applications that are sensitive to TCP's initial connection setup delay. To benefit from TFO, the first application data unit (e.g., an HTTP request) needs to be no more than TCP's maximum segment size (minus options used in SYN). Otherwise the remote server can only process the client's application data unit once the rest of it is delivered after the initial handshake, diminishing TFO's benefit.

To the extent possible, applications SHOULD reuse the connection to take advantage of TCP's built-in congestion control and reduce connection setup overhead. An application that employs too many short-lived connections will negatively impact network stability, as these connections often exit before TCP's congestion control algorithm takes effect.

6.3. Example: Web Clients and Servers

6.3.1. HTTP Request Replay

While TFO is motivated by Web applications, the browser should not use TFO to send requests in SYNs if those requests cannot tolerate replays. One example is POST requests without application-layer transaction protection (e.g., a unique identifier in the request header).

On the other hand, TFO is particularly useful for GET requests. GET requests replay could happen across striped TCP connections: after a server receives an HTTP request but before the ACKs of the requests reach the browser, the browser may timeout and retry the same request on another (possibly new) TCP connection. This differs from a TFO replay only in that the replay is initiated by the browser, not by the TCP stack.

6.3.2. HTTP over TLS (HTTPS)

For TLS over TCP, it is safe and useful to include TLS CLIENT_HELLO in the SYN packet to save one RTT in TLS handshake. There is no concern about violating idem-potency. In particular it can be used alone with the speculative connection above.

6.3.3. Comparison with HTTP Persistent Connections

Is TFO useful given the wide deployment of HTTP persistent connections? The short answer is yes. Studies [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 idle connections up to several minutes, well into "human think" time.

Keeping connections open and idle even longer risks a greater performance penalty. [HNESSK10][MQXMZ11] show that the majority of home routers and ISPs fail to meet the 124-minute idle timeout mandated in [RFC5382]. In [MQXMZ11], 35% of mobile ISPs silently timeout idle connections within 30 minutes. End hosts, unaware of silent middle-box timeouts, suffer multi-minute TCP timeouts upon using those long-idle connections.

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 such a prominent issue in modern LTE devices that mobile browsers close HTTP connections within seconds or even immediately [SOUDERS11].

[RCCJR11] studied Chrome browser [Chrome] performance based on 28 days of global statistics. The Chrome browser keeps idle HTTP persistent connections for 5 to 10 minutes. However the average number of the transactions per connection is only 3.3 and TCP 3WSH accounts for up to 25% of the HTTP transaction network latency. The authors estimated that TFO improves page load time by 10% to 40% on selected popular Web sites.

6.3.4. Load Balancers and Server farms

Servers behind a load balancers that accept connection requests to the same server IP address should use the same key such that they generate identical Fast Open Cookies for a particular client IP address. Otherwise a client may get different cookies across connections; its Fast Open attempts would fall back to regular 3WSH.

7. Open Areas for Experimentation

We now outline some areas that need experimentation in the Internet and under different network scenarios. These experiments should help

the community evaluate Fast Open benefits and risks towards further standardization and implementation of Fast Open and its related protocols.

7.1. Performance impact due to middle-boxes and NAT

[MAF04] found that some middle-boxes and end-hosts may drop packets with unknown TCP options. 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 falling back to regular TCP handshake and re-transmitting SYN without data or cookie options after the initial SYN timeout. Moreover the implementation is recommended to negatively cache such incidents to avoid recurring timeouts. Further study is required to evaluate the performance impact of these drop behaviors.

Another interesting study is the loss of TFO performance benefit behind certain carrier-grade NAT. Typically hosts behind a NAT sharing the same IP address will get the same cookie for 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 benefits. However, there is no performance penalty either, as described in Section "Client: Receiving SYN-ACK".

7.2. Impact on congestion control

Although TFO does not directly change the congestion control, there are subtle cases that it may. When SYN-ACK times out, regular TCP reduces the initial congestion window before sending any data [RFC5681]. However in TFO the server may have already sent up to an initial window of data.

If the server serves mostly short connections then the losses of SYN-ACKs are not as effective as regular TCP on reducing the congestion window. This could result in an unstable network condition. The connections that experience losses may attempt again and add more load under congestion. A potential solution is to temporarily disable Fast Open if the server observes many SYN-ACK or data losses during the handshake across connections. Further experimentation regarding the congestion control impact will be useful.

7.3. Cookie-less Fast Open

The cookie mechanism mitigates resource exhaustion and amplification attacks. However cookies are not necessary if the server has application-level protection or is immune to these attacks. For example a Web server that only replies with a simple HTTP redirect

response that fits in the SYN-ACK packet may not care about resource exhaustion.

For such applications the server may choose to generate a trivial or even a zero-length cookie to improve performance by avoiding the cookie generation and verification. If the server believes it's under a DoS attack through other defense mechanisms, it can switch to regular Fast Open for listener sockets.

8. Related Work

8.1. 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 3WS [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.

8.2. Common Defenses Against SYN Flood Attacks

[RFC4987] studies on mitigating attacks from regular SYN flood, i.e., SYN without data. 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.

8.3. Speculative Connections by the Applications

Some Web browsers maintain a history of the domains for frequently visited web pages. The browsers then speculatively pre-open TCP connections to these domains before the user initiates any requests for them [BELSHE11]. While this technique also saves the handshake latency, it wastes server and network resources by initiating and maintaining idle connections.

8.4. Fast Open Cookie in FIN

An alternate proposal is to request a TFO cookie in the FIN instead, since FIN-drop by incompatible middle-boxes does not affect latency. However paths that block SYN cookies may be more likely to drop a later SYN packet with data, and many applications close a connection with RST instead anyway.

Although cookie-in-FIN may not improve robustness, it would give clients using a single connection a latency advantage over clients opening multiple parallel connections. If experiments with TFO find that it leads to increased connection-sharding, cookie-in-FIN may prove to be a useful alternative.

8.5. 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. But the server can only send up to MSS bytes of data during the handshake instead of the initial congestion window unlike TFO. Therefore the latency of applications such as Web may be worse than with TFO.

9. IANA Considerations

IANA is requested to allocate one value from the TCP Option Kind Numbers: The constant-TBD in Section 4.1.1 has to be replaced with the newly assigned value. The length of the new TCP option Kind is variable and the Meaning should be set to "TCP Fast Open Cookie". Early implementation before the IANA allocation SHOULD follow [RFC6994] and use experimental option 254 and magic number 0xF989 (16 bits), then migrate to the new option after the allocation accordingly.

10. Acknowledgement

We thank Bob Briscoe, Michael Scharf, Gorrry Fairhurst, Rick Jones, Roberto Peon, William Chan, Adam Langley, Neal Cardwell, Eric Dumazet, and Matt Mathis for their feedbacks. We especially thank Barath Raghavan for his contribution on the security design of Fast Open and proofreading this draft numerous times.

11. References

11.1. 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.

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [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
- [RFC6994] Touch, Joe, "Shared Use of Experimental TCP Options", RFC6994, August 2013.
- [RFC3390] Allman, M., Floyd, S., and C. Partridge, "Increasing TCP's Initial Window", RFC 3390, October 2002.

11.2. Informative References

- [AERG11] Al-Fares, M., Elmeleegy, K., Reed, B., Gashinsky, I., "Overclocking the Yahoo! CDN for Faster Web Page Loads". In Proceedings of Internet Measurement Conference, November 2011.
- [HNESSK10] Haetoeenen, S., Nyrhinen, A., Eggert, L., Strowes, S., Sarolahti, P., Kojo., M., "An Experimental Study of Home Gateway Characteristics". In Proceedings of Internet Measurement Conference. October 2010
- [HNRGHT11] Honda, M., Nishida, Y., Raiciu, C., Greenhalgh, A., Handley, M., Tokuda, H., "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] Wang, Z., Qian, Z., Xu, Q., Mao, Z., Zhang, M., "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>

- [RCCJR11] Radhakrishnan, S., Cheng, Y., Chu, J., Jain, A., Raghavan, B., "TCP Fast Open". In Proceedings of 7th ACM CoNEXT Conference, December 2011.
- [RFC1323] Jacobson, V., Braden, R., Borman, D., "TCP Extensions for High Performance", RFC 1323, May 1992.
- [RFC1644] Braden, R., "T/TCP -- TCP Extensions for Transactions Functional Specification", RFC 1644, July 1994.
- [RFC2460] Deering, S., Hinden, R., "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, December 1998.
- [RFC4987] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", RFC 4987, August 2007.
- [RFC6013] Simpson, W., "TCP Cookie Transactions (TCPCT)", RFC6013, January 2011.
- [SOUDERS11] Souders, S., "Making A Mobile Connection". <http://www.stevesouders.com/blog/2011/09/21/making-a-mobile-connection/>
- [BRISCOE12] Briscoe, B., "Some ideas building on draft-ietf-tcpm-fastopen-01", tcpm list, <http://www.ietf.org/mail-archive/web/tcpm/current/msg07192.html>
- [BELSHE11] Belshe, M., "The era of browser preconnect.", <http://www.belshe.com/2011/02/10/the-era-of-browser-preconnect/>
- [JIDKT07] Jaiswal, S., Iannaccone, G., Diot, C., Kurose, J., Towsley, D., "Measurement and classification of out-of-sequence packets in a tier-1 IP backbone". IEEE/ACM Transactions on Networking (TON), 15(1), 54-66.
- [Chrome] Chrome. <https://www.google.com/intl/en-US/chrome/browser/>

Appendix A. Example Socket API Changes to support TFO

A.1 Active Open

The active open side involves changing or replacing the `connect()` call, which does not take a user data buffer argument. We recommend replacing `connect()` call to minimize API changes and hence

applications to reduce the deployment hurdle.

One solution implemented in Linux 3.7 is introducing a new flag `MSG_FASTOPEN` for `sendto()` or `sendmsg()`. `MSG_FASTOPEN` marks the attempt to send data in SYN like a combination of `connect()` and `sendto()`, by performing an implicit `connect()` operation. It blocks until the handshake has completed and the data is buffered.

For non-blocking socket it returns the number of bytes buffered and sent in the SYN packet. If the cookie is not available locally, it returns `-1` with `errno EINPROGRESS`, and sends a SYN with TFO cookie request automatically. The caller needs to write the data again when the socket is connected. On errors, it returns the same `errno` as `connect()` if the handshake fails.

An implementation may prefer not to change the `sendmsg()` because TFO is a TCP specific feature. A solution is to add a new socket option `TCP_FASTOPEN` for TCP sockets. When the option is enabled before a `connect` operation, `sendmsg()` or `sendto()` will perform Fast Open operation similar to the `MSG_FASTOPEN` flag described above. This approach however requires an extra `setsockopt()` system call.

A.2 Passive Open

The passive open side change is simpler compared to active open side. The application only needs to enable the reception of Fast Open requests via a new `TCP_FASTOPEN` `setsockopt()` socket option before `listen()`.

The option enables Fast Open on the listener socket. The option value specifies the `PendingFastOpenRequests` threshold, i.e., the maximum length of pending SYNs with data payload. Once enabled, the TCP implementation will respond with TFO cookies per request.

Traditionally `accept()` returns only after a socket is connected. But for a Fast Open connection, `accept()` returns upon receiving a SYN with a valid Fast Open cookie and data, and the data is available to be read through, e.g., `recvmsg()`, `read()`.

Authors' 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

TCP Maintenance and Minor Extensions
(tcpm)
Internet-Draft
Intended status: Informational
Expires: April 18, 2013

M. Kuehlewind, Ed.
University of Stuttgart
R. Scheffenegger
NetApp, Inc.
October 15, 2012

Problem Statement and Requirements for a More Accurate ECN Feedback
draft-kuehlewind-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

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

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

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

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

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of

publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

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
Header Length				Reserved			N S	C W R	E C E	U R G	A C K	P S H	R S T	S Y N	F I 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 approach 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

TCP Maintenance & Minor Extensions (tcpm)
Internet-Draft
Intended status: Experimental
Expires: April 21, 2016

B. Briscoe
Simula Research Laboratory
M. Kuehlewind
ETH Zurich
R. Scheffenegger
NetApp, Inc.
October 19, 2015

More Accurate ECN Feedback in TCP
draft-kuehlewind-tcpm-accurate-ecn-05

Abstract

Explicit Congestion Notification (ECN) is a mechanism where network nodes can mark IP packets instead of dropping them to indicate incipient congestion to the end-points. Receivers with an ECN-capable transport protocol feed back 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 Congestion Exposure (ConEx) or Data Center TCP (DCTCP) need more accurate ECN feedback information whenever more than one marking is received in one RTT. This document specifies an experimental scheme to provide more than one feedback signal per RTT in the TCP header. Given TCP header space is scarce, it overloads the three existing ECN-related flags in the TCP header and provides additional information in a new TCP option.

Status of This Memo

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

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

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

This Internet-Draft will expire on April 21, 2016.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Document Roadmap	4
1.2. Goals	4
1.3. Experiment Goals	5
1.4. Terminology	5
1.5. Recap of Existing ECN feedback in IP/TCP	6
2. AccECN Protocol Overview and Rationale	7
2.1. Capability Negotiation	8
2.2. Feedback Mechanism	8
2.3. Delayed ACKs and Resilience Against ACK Loss	9
2.4. Feedback Metrics	10
2.5. Generic (Dumb) Reflector	10
3. AccECN Protocol Specification	11
3.1. Negotiation during the TCP handshake	11
3.2. AccECN Feedback	14
3.2.1. The ACE Field	14
3.2.2. Safety against Ambiguity of the ACE Field	16
3.2.3. The AccECN Option	16
3.2.4. Path Traversal of the AccECN Option	17
3.2.5. Usage of the AccECN TCP Option	19
3.3. AccECN Compliance by TCP Proxies, Offload Engines and other Middleboxes	20
4. Interaction with Other TCP Variants	21
4.1. Compatibility with SYN Cookies	21
4.2. Compatibility with Other TCP Options and Experiments	21
4.3. Compatibility with Feedback Integrity Mechanisms	21
5. Protocol Properties	23
6. IANA Considerations	25
7. Security Considerations	25
8. Acknowledgements	26
9. Comments Solicited	26

10. References	26
10.1. Normative References	26
10.2. Informative References	27
Appendix A. Example Algorithms	29
A.1. Example Algorithm to Encode/Decode the AcceECN Option . .	29
A.2. Example Algorithm for Safety Against Long Sequences of ACK Loss	30
A.2.1. Safety Algorithm without the AcceECN Option	30
A.2.2. Safety Algorithm with the AcceECN Option	32
A.3. Example Algorithm to Estimate Marked Bytes from Marked Packets	33
A.4. Example Algorithm to Beacon AcceECN Options	34
A.5. Example Algorithm to Count Not-ECT Bytes	35
Appendix B. Alternative Design Choices (To Be Removed Before Publication)	35
Appendix C. Open Protocol Design Issues (To Be Removed Before Publication)	36
Appendix D. Changes in This Version (To Be Removed Before Publication)	37
Authors' Addresses	37

1. Introduction

Explicit Congestion Notification (ECN) [RFC3168] is a mechanism where network nodes can mark IP packets instead of dropping them to indicate incipient congestion to the end-points. Receivers with an ECN-capable transport protocol feed back 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 [I-D.ietf-conex-abstract-mech]) or DCTCP [I-D.bensley-tcpm-dctcp] need more accurate ECN feedback information whenever more than one marking is received in one RTT. A fuller treatment of the motivation for this specification is given in the associated requirements document [RFC7560].

This documents specifies an experimental scheme for ECN feedback in the TCP header to provide more than one feedback signal per RTT. It will be called the more accurate ECN feedback scheme, or AcceECN for short. If AcceECN progresses from experimental to the standards track, it is intended to be a complete replacement for classic ECN feedback, not a fork in the design of TCP. Thus, the applicability of AcceECN is intended to include all public and private IP networks (and even any non-IP networks over which TCP is used today). Until the AcceECN experiment succeeds, [RFC3168] will remain as the standards track specification for adding ECN to TCP. To avoid confusion, in this document we use the term 'classic ECN' for the pre-existing ECN specification [RFC3168].

AcceECN is solely an (experimental) change to the TCP wire protocol. It is completely independent of how TCP might respond to congestion feedback. This specification overloads flags and fields in the main TCP header with new definitions, so both ends have to support the new wire protocol before it can be used. Therefore during the TCP handshake the two ends use the three ECN-related flags in the TCP header to negotiate the most advanced feedback protocol that they can both support.

It is likely (but not required) that the AcceECN protocol will be implemented along with the following experimental additions to the TCP-ECN protocol: ECN-capable SYN/ACK [RFC5562], ECN path-probing and fall-back [I-D.kuehlewind-tcpm-ecn-fallback] and testing receiver non-compliance [I-D.moncaster-tcpm-rcv-cheat].

1.1. Document Roadmap

The following introductory sections outline the goals of AcceECN (Section 1.2) and the goal of experiments with ECN (Section 1.3) so that it is clear what success would look like. Then terminology is defined (Section 1.4) and a recap of existing prerequisite technology is given (Section 1.5).

Section 2 gives an informative overview of the AcceECN protocol. Then Section 3 gives the normative protocol specification. Section 4 assesses the interaction of AcceECN with commonly used variants of TCP, whether standardised or not. Section 5 summarises the features and properties of AcceECN.

Section 6 summarises the protocol fields and numbers that IANA will need to assign and Section 7 points to the aspects of the protocol that will be of interest to the security community.

Appendix A gives pseudocode examples for the various algorithms that AcceECN uses.

1.2. Goals

[RFC7560] enumerates requirements that a candidate feedback scheme will need to satisfy, under the headings: resilience, timeliness, integrity, accuracy (including ordering and lack of bias), complexity, overhead and compatibility (both backward and forward). It recognises that a perfect scheme that fully satisfies all the requirements is unlikely and trade-offs between requirements are likely. Section 5 presents the properties of AcceECN against these requirements and discusses the trade-offs made.

The requirements document recognises that a protocol as ubiquitous as TCP needs to be able to serve as-yet-unspecified requirements. Therefore an AccECN receiver aims to act as a generic (dumb) reflector of congestion information so that in future new sender behaviours can be deployed unilaterally.

1.3. Experiment Goals

TCP is critical to the robust functioning of the Internet, therefore any proposed modifications to TCP need to be thoroughly tested. The present specification describes an experimental protocol that adds more accurate ECN feedback to the TCP protocol. The intention is to specify the protocol sufficiently so that more than one implementation can be built in order to test its function, robustness and interoperability (with itself and with previous version of ECN and TCP).

The experimental protocol will be considered successful if it satisfies the requirements of [RFC7560] in the consensus opinion of the IETF tcpm working group. In short, this requires that it improves the accuracy and timeliness of TCP's ECN feedback, as claimed in Section 5, while striking a balance between the conflicting requirements of resilience, integrity and minimisation of overhead. It also requires that it is not unduly complex, and that it is compatible with prevalent equipment behaviours in the current Internet, whether or not they comply with standards.

1.4. Terminology

AccECN: The more accurate ECN feedback scheme will be called AccECN for short.

Classic ECN: the ECN protocol specified in [RFC3168].

Classic ECN feedback: the feedback aspect of the ECN protocol specified in [RFC3168], including generation, encoding, transmission and decoding of feedback, but not the Data Sender's subsequent response to that feedback.

ACK: A TCP acknowledgement, with or without a data payload.

Pure ACK: A TCP acknowledgement without a data payload.

TCP client: The TCP stack that originates a connection.

TCP server: The TCP stack that responds to a connection request.

Data Receiver: The endpoint of a TCP half-connection that receives data and sends AccECN feedback.

Data Sender: The endpoint of a TCP half-connection that sends data and receives AccECN feedback.

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].

1.5. Recap of Existing ECN feedback in IP/TCP

ECN [RFC3168] uses two bits in the IP header. Once ECN has been negotiated with the receiver at the transport layer, an ECN sender can set two possible codepoints (ECT(0) or ECT(1)) in the IP header to indicate an ECN-capable transport (ECT). If both ECN bits are zero, the packet is considered to have been sent by a Not-ECN-capable Transport (Not-ECT). When a network node experiences congestion, it will occasionally either drop or mark a packet, with the choice depending on the packet's ECN codepoint. If the codepoint is Not-ECT, only drop is appropriate. If the codepoint is ECT(0) or ECT(1), the node can mark the packet by setting both ECN bits, which is termed 'Congestion Experienced' (CE), or loosely a 'congestion mark'. Table 1 summarises these codepoints.

IP-ECN codepoint (binary)	Codepoint name	Description
00	Not-ECT	Not ECN-Capable Transport
01	ECT(1)	ECN-Capable Transport (1)
10	ECT(0)	ECN-Capable Transport (0)
11	CE	Congestion Experienced

Table 1: The ECN Field in the IP Header

In the TCP header the first two bits in byte 14 are defined as flags for the use of ECN (CWR and ECE in Figure 1 [RFC3168]). A TCP client indicates it supports ECN by setting ECE=CWR=1 in the SYN, and an ECN-enabled server confirms ECN support by setting ECE=1 and CWR=0 in the SYN/ACK. On reception of a CE-marked packet at the IP layer, the Data Receiver starts to set the Echo Congestion Experienced (ECE) flag continuously in the TCP header of ACKs, which ensures the signal is received reliably even if ACKs are lost. The TCP sender confirms that it has received at least one ECE signal by responding with the congestion window reduced (CWR) flag, which allows the TCP receiver to stop repeating the ECN-Echo flag. This always leads to a full RTT

of ACKs with ECE set. Thus any additional CE markings arriving within this RTT cannot be fed back.

The ECN Nonce [RFC3540] is an optional experimental addition to ECN that the TCP sender can use to protect against accidental or malicious concealment of marked or dropped packets. The sender can send an ECN nonce, which is a continuous pseudo-random pattern of ECT(0) and ECT(1) codepoints in the ECN field. The receiver is required to feed back a 1-bit nonce sum that counts the occurrence of ECT(1) packets using the last bit of byte 13 in the TCP header, which is defined as the Nonce Sum (NS) flag.

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

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

2. AccECN Protocol Overview and Rationale

This section provides an informative overview of the AccECN protocol that will be normatively specified in Section 3

Like the original TCP approach, the Data Receiver of each TCP half-connection sends AccECN feedback to the Data Sender on TCP acknowledgements, reusing data packets of the other half-connection whenever possible.

The AccECN protocol has had to be designed in two parts:

- o an essential part that re-uses ECN TCP header bits to feed back the number of arriving CE marked packets. This provides more accuracy than classic ECN feedback, but limited resilience against ACK loss;
- o a supplementary part using a new AccECN TCP Option that provides additional feedback on the number of bytes that arrive marked with each of the three ECN codepoints (not just CE marks). This provides greater resilience against ACK loss than the essential feedback, but it is more likely to suffer from middlebox interference.

The two part design was necessary, given limitations on the space available for TCP options and given the possibility that certain incorrectly designed middleboxes prevent TCP using any new options.

The essential part overloads the previous definition of the three flags in the TCP header that had been assigned for use by ECN. This design choice deliberately replaces the classic ECN feedback protocol, rather than leaving classic ECN feedback intact and adding more accurate feedback separately because:

- o this efficiently reuses scarce TCP header space, given TCP option space is approaching saturation;
- o a single upgrade path for the TCP protocol is preferable to a fork in the design;
- o otherwise classic and accurate ECN feedback could give conflicting feedback on the same segment, which could open up new security concerns and make implementations unnecessarily complex;
- o middleboxes are more likely to faithfully forward the TCP ECN flags than newly defined areas of the TCP header.

AccECN is designed to work even if the supplementary part is removed or zeroed out, as long as the essential part gets through.

2.1. Capability Negotiation

AccECN is a change to the wire protocol of the main TCP header, therefore it can only be used if both endpoints have been upgraded to understand it. The TCP client signals support for AccECN on the initial SYN of a connection and the TCP server signals whether it supports AccECN on the SYN/ACK. The TCP flags on the SYN that the client uses to signal AccECN support have been carefully chosen so that a TCP server will interpret them as a request to support the most recent variant of ECN feedback that it supports. Then the client falls back to the same variant of ECN feedback.

An AccECN TCP client does not send the new AccECN Option on the SYN as SYN option space is limited and successful negotiation using the flags in the main header is taken as sufficient evidence that both ends also support the AccECN Option. The TCP server sends the AccECN Option on the SYN/ACK and the client sends it on the first ACK to test whether the network path forwards the option correctly.

2.2. Feedback Mechanism

A Data Receiver maintains four counters initialised at the start of the half-connection. Three count the number of arriving payload bytes marked CE, ECT(1) and ECT(0) respectively. The fourth counts the number of packets arriving marked with a CE codepoint (including control packets without payload if they are CE-marked).

The Data Sender maintains four equivalent counters for the half connection, and the AccECN protocol is designed to ensure they will match the values in the Data Receiver's counters, albeit after a little delay.

Each ACK carries the three least significant bits (LSBs) of the packet-based CE counter using the ECN bits in the TCP header, now renamed the Accurate ECN (ACE) field. The LSBs of each of the three byte counters are carried in the AccECN Option.

2.3. Delayed ACKs and Resilience Against ACK Loss

With both the ACE and the AccECN Option mechanisms, the Data Receiver continually repeats the current LSBs of each of its respective counters. Then, even if some ACKs are lost, the Data Sender should be able to infer how much to increment its own counters, even if the protocol field has wrapped.

The 3-bit ACE field can wrap fairly frequently. Therefore, even if it appears to have incremented by one (say), the field might have actually cycled completely then incremented by one. The Data Receiver is required not to delay sending an ACK to such an extent that the ACE field would cycle. However cycling is still a possibility at the Data Sender because a whole sequence of ACKs carrying intervening values of the field might all be lost or delayed in transit.

The fields in the AccECN Option are larger, but they will increment in larger steps because they count bytes not packets. Nonetheless, their size has been chosen such that a whole cycle of the field would never occur between ACKs unless there had been an infeasibly long sequence of ACK losses. Therefore, as long as the AccECN Option is available, it can be treated as a dependable feedback channel.

If the AccECN Option is not available, e.g. it is being stripped by a middlebox, the AccECN protocol will only feed back information on CE markings (using the ACE field). Although not ideal, this will be sufficient, because it is envisaged that neither ECT(0) nor ECT(1) will ever indicate more severe congestion than CE, even though future uses for ECT(0) or ECT(1) are still unclear. Because the 3-bit ACE field is so small, when it is the only field available the Data Sender has to interpret it conservatively assuming the worst possible wrap.

Certain specified events trigger the Data Receiver to include an AccECN Option on an ACK. The rules are designed to ensure that the order in which different markings arrive at the receiver is communicated to the sender (as long as there is no ACK loss).

Implementations are encouraged to send an AcceECN Option more frequently, but this is left up to the implementer.

2.4. Feedback Metrics

The CE packet counter in the ACE field and the CE byte counter in the AcceECN Option both provide feedback on received CE-marks. The CE packet counter includes control packets that do not have payload data, while the CE byte counter solely includes marked payload bytes. If both are present, the byte counter in the option will provide the more accurate information needed for modern congestion control and policing schemes, such as DCTCP or ConEx. If the option is stripped, a simple algorithm to estimate the number of marked bytes from the ACE field is given in Appendix A.3.

Feedback in bytes is recommended in order to protect against the receiver using attacks similar to 'ACK-Division' to artificially inflate the congestion window, which is why [RFC5681] now recommends that TCP counts acknowledged bytes not packets.

2.5. Generic (Dumb) Reflector

The ACE field provides information about CE markings on both data and control packets. According to [RFC3168] the Data Sender is meant to set control packets to Not-ECT. However, mechanisms in certain private networks (e.g. data centres) set control packets to be ECN capable because they are precisely the packets that performance depends on most.

For this reason, AcceECN is designed to be a generic reflector of whatever ECN markings it sees, whether or not they are compliant with a current standard. Then as standards evolve, Data Senders can upgrade unilaterally without any need for receivers to upgrade too. It is also useful to be able to rely on generic reflection behaviour when senders need to test for unexpected interference with markings (for instance [I-D.kuehlewind-tcpm-ecn-fallback] and [I-D.moncaster-tcpm-rcv-cheat]).

The initial SYN is the most critical control packet, so AcceECN provides feedback on whether it is CE marked, even though it is not allowed to be ECN-capable according to RFC 3168. However, middleboxes have been known to overwrite the ECN IP field as if it is still part of the old Type of Service (ToS) field. If a TCP client has set the SYN to Not-ECT, but receives CE feedback, it can detect such middlebox interference and send Not-ECT for the rest of the connection (see [I-D.kuehlewind-tcpm-ecn-fallback] for the detailed fall-back behaviour).

Today, if a TCP server receives CE on a SYN, it cannot know whether it is invalid (or valid) because only the TCP client knows whether it originally marked the SYN as Not-ECT (or ECT). Therefore, the server's only safe course of action is to disable ECN for the connection. Instead, the AccECN protocol allows the server to feed back the CE marking to the client, which then has all the information to decide whether the connection has to fall-back from supporting ECN (or not).

Providing feedback of CE marking on the SYN also supports future scenarios in which SYNs might be ECN-enabled (without prejudging whether they ought to be). For instance, in certain environments such as data centres, it might be appropriate to allow ECN-capable SYNs. Then, if feedback showed the SYN had been CE marked, the TCP client could reduce its initial window (IW). It could also reduce IW conservatively if feedback showed the receiver did not support ECN (because if there had been a CE marking, the receiver would not have understood it). Note that this text merely motivates dumb reflection of CE on a SYN, it does not judge whether a SYN ought to be ECN-capable.

3. AccECN Protocol Specification

3.1. Negotiation during the TCP handshake

During the TCP handshake at the start of a connection, to request more accurate ECN feedback the TCP client (host A) MUST set the TCP flags NS=1, CWR=1 and ECE=1 in the initial SYN segment.

If a TCP server (B) that is AccECN enabled receives a SYN with the above three flags set, it MUST set both its half connections into AccECN mode. Then it MUST set the flags CWR=1 and ECE=0 on its response in the SYN/ACK segment to confirm that it supports AccECN. The TCP server MUST NOT set this combination of flags unless the preceding SYN requested support for AccECN as above.

A TCP server in AccECN mode MUST additionally set the flag NS=1 on the SYN/ACK if the SYN was CE-marked (see Section 2.5). If the received SYN was Not-ECT, ECT(0) or ECT(1), it MUST clear NS (NS=0) on the SYN/ACK.

Once a TCP client (A) has sent the above SYN to declare that it supports AccECN, and once it has received the above SYN/ACK segment that confirms that the TCP server supports AccECN, the TCP client MUST set both its half connections into AccECN mode.

If after the normal TCP timeout the TCP client has not received a SYN/ACK to acknowledge its SYN, the SYN might just have been lost,

e.g. due to congestion, or a middlebox might be blocking segments with the AccECN flags. To expedite connection setup, the host SHOULD fall back to NS=CWR=ECE=0 on the retransmission of the SYN. It would make sense to also remove any other experimental fields or options on the SYN in case a middlebox might be blocking them, although the required behaviour will depend on the specification of the other option(s) and any attempt to co-ordinate fall-back between different modules of the stack. Implementers MAY use other fall-back strategies if they are found to be more effective (e.g. attempting to retransmit a second AccECN segment before fall-back, falling back to classic ECN feedback rather than non-ECN, and/or caching the result of a previous attempt to access the same host while negotiating AccECN).

The fall-back procedure if the TCP server receives no ACK to acknowledge a SYN/ACK that tried to negotiate AccECN is specified in Section 3.2.4.

The three flags set to 1 to indicate AccECN support on the SYN have been carefully chosen to enable natural fall-back to prior stages in the evolution of ECN. Table 2 tabulates all the negotiation possibilities for ECN-related capabilities that involve at least one AccECN-capable host. To compress the width of the table, the headings of the first four columns have been severely abbreviated, as follows:

Ac: More *Ac*curate ECN Feedback

N: ECN-*N*once [RFC3540]

E: *E*CN [RFC3168]

I: Not-ECN (*I*mplicit congestion notification using packet drop).

Ac	N	E	I	SYN A->B			SYN/ACK B->A			Feedback Mode
AB				NS	CWR	ECE	NS	CWR	ECE	AccECN
AB				1	1	1	0	1	0	AccECN (CE on SYN)
A	B			1	1	1	1	0	1	classic ECN
A		B		1	1	1	0	0	1	classic ECN
A			B	1	1	1	0	0	0	Not ECN
B	A			0	1	1	0	0	1	classic ECN
B		A		0	1	1	0	0	1	classic ECN
B			A	0	0	0	0	0	0	Not ECN
A			B	1	1	1	1	1	1	Not ECN (broken)
A				1	1	1	0	1	1	Not ECN (see Appx B)
A				1	1	1	1	0	0	Not ECN (see Appx B)

Table 2: ECN capability negotiation between Originator (A) and Responder (B)

Table 2 is divided into blocks each separated by an empty row.

1. The top block shows the case already described where both endpoints support AccECN and how the TCP server (B) indicates congestion feedback.
2. The second block shows the cases where the TCP client (A) supports AccECN but the TCP server (B) supports some earlier variant of TCP feedback, indicated in its SYN/ACK. Therefore, as soon as an AccECN-capable TCP client (A) receives the SYN/ACK shown it MUST set both its half connections into the feedback mode shown in the rightmost column.
3. The third block shows the cases where the TCP server (B) supports AccECN but the TCP client (A) supports some earlier variant of TCP feedback, indicated in its SYN. Therefore, as soon as an AccECN-enabled TCP server (B) receives the SYN shown, it MUST set both its half connections into the feedback mode shown in the rightmost column.
4. The fourth block displays combinations that are not valid or currently unused and therefore both ends MUST fall-back to Not ECN for both half connections. Especially the first case (marked 'broken') where all bits set in the SYN are reflected by the receiver in the SYN/ACK, which happens quite often if the TCP

connection is proxied.{ToDo: Consider using the last two cases for AccECN f/b of ECT(0) and ECT(1) on the SYN (Appendix B)}

The following exceptional cases need some explanation:

ECN Nonce: An AccECN implementation, whether client or server, sender or receiver, does not need to implement the ECN Nonce behaviour [RFC3540]. AccECN is compatible with an alternative ECN feedback integrity approach that does not use up the ECT(1) codepoint and can be implemented solely at the sender (see Section 4.3).

Simultaneous Open: An originating AccECN Host (A), having sent a SYN with NS=1, CWR=1 and ECE=1, might receive another SYN from host B. Host A MUST then enter the same feedback mode as it would have entered had it been a responding host and received the same SYN. Then host A MUST send the same SYN/ACK as it would have sent had it been a responding host (see the third block above).

3.2. AccECN Feedback

Each Data Receiver maintains four counters, `r.cep`, `r.ceb`, `r.e0b` and `r.elb`. The CE packet counter (`r.cep`), counts the number of packets the host receives with the CE code point in the IP ECN field, including CE marks on control packets without data. `r.ceb`, `r.e0b` and `r.elb` count the number of TCP payload bytes in packets marked respectively with the CE, ECT(0) and ECT(1) codepoint in their IP-ECN field. When a host first enters AccECN mode, it initialises its counters to `r.cep = 6`, `r.e0b = 1` and `r.ceb = r.elb = 0` (see Appendix A.5). Non-zero initial values are used to be distinct from cases where the fields are incorrectly zeroed (e.g. by middleboxes).

A host feeds back the CE packet counter using the Accurate ECN (ACE) field, as explained in the next section. And it feeds back all the byte counters using the AccECN TCP Option, as specified in Section 3.2.3. Whenever a host feeds back the value of any counter, it MUST report the most recent value, no matter whether it is in a pure ACK, an ACK with new payload data or a retransmission.

3.2.1. The ACE Field

After AccECN has been negotiated on the SYN and SYN/ACK, both hosts overload the three TCP flags ECE, CWR and NS in the main TCP header as one 3-bit field. Then the field is given a new name, ACE, as shown in Figure 2.

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

Figure 2: Definition of the ACE field within bytes 13 and 14 of the TCP Header (when AccECN has been negotiated and SYN=0).

The original definition of these three flags in the TCP header, including the addition of support for the ECN Nonce, is shown for comparison in Figure 1. This specification does not rename these three TCP flags, it merely overloads them with another name and definition once an AccECN connection has been established.

A host **MUST** interpret the ECE, CWR and NS flags as the 3-bit ACE counter on a segment with SYN=0 that it sends or receives if both of its half-connections are set into AccECN mode having successfully negotiated AccECN (see Section 3.1). A host **MUST NOT** interpret the 3 flags as a 3-bit ACE field on any segment with SYN=1 (whether ACK is 0 or 1), or if AccECN negotiation is incomplete or has not succeeded.

Both parts of each of these conditions are equally important. For instance, even if AccECN negotiation has been successful, the ACE field is not defined on any segments with SYN=1 (e.g. a retransmission of an unacknowledged SYN/ACK, or when both ends send SYN/ACKs after AccECN support has been successfully negotiated during a simultaneous open).

The ACE field encodes the three least significant bits of the r.cep counter, therefore its initial value will be 0b110 (decimal 6). This non-zero initialization allows a TCP server to use a stateless handshake (see Section 4.1) but still detect from the TCP client's first ACK that the client considers it has successfully negotiated AccECN. If the SYN/ACK was CE marked, the client **MUST** increase its r.cep counter before it sends its first ACK, therefore the initial value of the ACE field will be 0b111 (decimal 7). These values have deliberately been chosen such that they are distinct from [RFC5562] behaviour, where the TCP client would set ECE on the first ACK as feedback for a CE mark on the SYN/ACK.

If the value of the ACE field on the first segment with SYN=0 in either direction is anything other than 0b110 or 0b111, the Data Receiver **MUST** disable ECN for the remainder of the half-connection by marking all subsequent packets as Not-ECT.

3.2.2. Safety against Ambiguity of the ACE Field

If too many CE-marked segments are acknowledged at once, or if a long run of ACKs is lost, the 3-bit counter in the ACE field might have cycled between two ACKs arriving at the Data Sender.

Therefore an AccECN Data Receiver SHOULD immediately send an ACK once 'n' CE marks have arrived since the previous ACK, where 'n' SHOULD be 2 and MUST be no greater than 6.

If the Data Sender has not received AccECN TCP Options to give it more dependable information, and it detects that the ACE field could have cycled under the prevailing conditions, it SHOULD conservatively assume that the counter did cycle. It can detect if the counter could have cycled by using the jump in the acknowledgement number since the last ACK to calculate or estimate how many segments could have been acknowledged. An example algorithm to implement this policy is given in Appendix A.2. An implementer MAY develop an alternative algorithm as long as it satisfies these requirements.

If missing acknowledgement numbers arrive later (reordering) and prove that the counter did not cycle, the Data Sender MAY attempt to neutralise the effect of any action it took based on a conservative assumption that it later found to be incorrect.

3.2.3. The AccECN Option

The AccECN Option is defined as shown below in Figure 3. It consists of three 24-bit fields that provide the 24 least significant bits of the r.e0b, r.ceb and r.elb counters, respectively. The initial 'E' of each field name stands for 'Echo'.

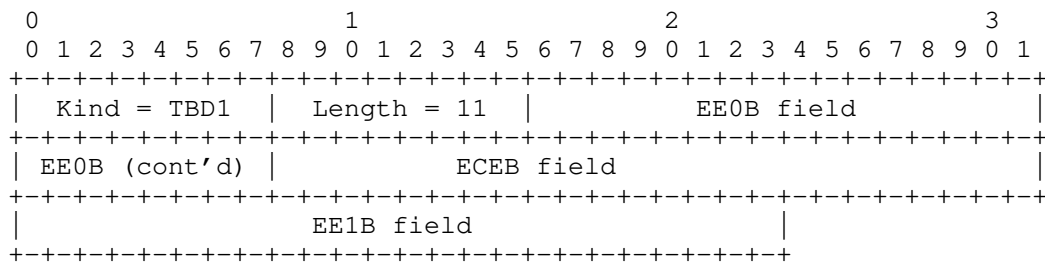


Figure 3: The AccECN Option

The Data Receiver MUST set the Kind field to TBD1, which is registered in Section 6 as a new TCP option Kind called AccECN. An experimental TCP option with Kind=254 MAY be used for initial experiments, with magic number 0xACCE.

Appendix A.1 gives an example algorithm for the Data Receiver to encode its byte counters into the AccECN Option, and for the Data Sender to decode the AccECN Option fields into its byte counters.

Note that there is no field to feedback Not-ECT bytes. Nonetheless an algorithm for the Data Sender to calculate the number of payload bytes received as Not-ECT is given in Appendix A.5.

Whenever a Data Receiver sends an AccECN Option, the rules in Section 3.2.5 expect it to always send a full-length option. To cope with option space limitations, it can omit unchanged fields from the tail of the option, as long as it preserves the order of the remaining fields and includes any field that has changed. The length field MUST indicate which fields are present as follows:

Length=11: EE0B, ECEB, EE1B

Length=8: EE0B, ECEB

Length=5: EE0B

Length=2: (empty)

The empty option of Length=2 is provided to allow for a case where an AccECN Option has to be sent (e.g. on the SYN/ACK to test the path), but there is very limited space for the option. For initial experiments, the Length field MUST be 2 greater to accommodate the 16-bit magic number.

All implementations of a Data Sender MUST be able to read in AccECN Options of any of the above lengths. They MUST ignore an AccECN Option of any other length.

3.2.4. Path Traversal of the AccECN Option

An AccECN host MUST NOT include the AccECN TCP Option on the SYN. Nonetheless, if the AccECN negotiation using the ECN flags in the main TCP header (Section 3.1) is successful, it implicitly declares that the endpoints also support the AccECN TCP Option.

If the TCP client indicated AccECN support, a TCP server that confirms its support for AccECN (as described in Section 3.1) SHOULD also include an AccECN TCP Option in the SYN/ACK. A TCP client that has successfully negotiated AccECN SHOULD include an AccECN Option in the first ACK at the end of the 3WSH. However, this first ACK is not delivered reliably, so the TCP client SHOULD also include an AccECN Option on the first data segment it sends (if it ever sends one). A host need not include an AccECN Option in any of these three cases if

it has cached knowledge that the packet would be likely to be blocked on the path to the other host if it included an AcceCN Option.

If the TCP client has successfully negotiated AcceCN but does not receive an AcceCN Option on the SYN/ACK, it switches into a mode that assumes that the AcceCN Option is not available for this half connection. Similarly, if the TCP server has successfully negotiated AcceCN but does not receive an AcceCN Option on the first ACK or on the first data segment, it switches into a mode that assumes that the AcceCN Option is not available for this half connection.

While a host is in the mode that assumes the AcceCN Option is not available, it MUST adopt the conservative interpretation of the ACE field discussed in Section 3.2.2. However, it cannot make any assumption about support of the AcceCN Option on the other half connection, so it MUST continue to send the AcceCN Option itself.

If after the normal TCP timeout the TCP server has not received an ACK to acknowledge its SYN/ACK, the SYN/ACK might just have been lost, e.g. due to congestion, or a middlebox might be blocking the AcceCN Option. To expedite connection setup, the host SHOULD fall back to NS=CWR=ECE=0 and no AcceCN Option on the retransmission of the SYN/ACK. Implementers MAY use other fall-back strategies if they are found to be more effective (e.g. retransmitting a SYN/ACK with AcceCN TCP flags but not the AcceCN Option; attempting to retransmit a second AcceCN segment before fall-back (most appropriate during high levels of congestion); or falling back to classic ECN feedback rather than non-ECN).

Similarly, if the TCP client detects that the first data segment it sent was lost, it SHOULD fall back to no AcceCN Option on the retransmission. Again, implementers MAY use other fall-back strategies such as attempting to retransmit a second segment with the AcceCN Option before fall-back, and/or caching the result of previous attempts.

Either host MAY include the AcceCN Option in a subsequent segment to retest whether the AcceCN Option can traverse the path.

Currently the Data Sender is not required to test whether the arriving byte counters in the AcceCN Option have been correctly initialised. This allows different initial values to be used as an additional signalling channel in future. If any inappropriate zeroing of these fields is discovered during testing, this approach will need to be reviewed.

3.2.5. Usage of the AccECN TCP Option

The following rules determine when a Data Receiver in AccECN mode sends the AccECN TCP Option, and which fields to include:

Change-Triggered ACKs: If an arriving packet increments a different byte counter to that incremented by the previous packet, the Data Receiver SHOULD immediately send an ACK with an AccECN Option, without waiting for the next delayed ACK. Certain offload hardware might not be able to support change-triggered ACKs, but otherwise it is important to keep exceptions to this rule to a minimum so that Data Senders can generally rely on this behaviour;

Continual Repetition: Otherwise, if arriving packets continue to increment the same byte counter, the Data Receiver can include an AccECN Option on most or all (delayed) ACKs, but it does not have to. If option space is limited on a particular ACK, the Data Receiver MUST give precedence to SACK information about loss. It SHOULD include an AccECN Option if the r.ceb counter has incremented and it MAY include an AccECN Option if r.ec0b or r.ec1b has incremented;

Full-Length Options Preferred: It SHOULD always use full-length AccECN Options. It MAY use shorter AccECN Options if space is limited, but it MUST include the counter(s) that have incremented since the previous AccECN Option and it MUST only truncate fields from the right-hand tail of the option to preserve the order of the remaining fields (see Section 3.2.3);

Beaconing Full-Length Options: Nonetheless, it MUST include a full-length AccECN TCP Option on at least three ACKs per RTT, or on all ACKs if there are less than three per RTT (see Appendix A.4 for an example algorithm that satisfies this requirement).

The following example series of arriving marks illustrates when a Data Receiver will emit an ACK if it is using a delayed ACK factor of 2 segments and change-triggered ACKs: 01 -> ACK, 01, 01 -> ACK, 10 -> ACK, 10, 01 -> ACK, 01, 11 -> ACK, 01 -> ACK.

For the avoidance of doubt, the change-triggered ACK mechanism ignores the arrival of a control packet with no payload, because it does not alter any byte counters. The change-triggered ACK approach will lead to some additional ACKs but it feeds back the timing and the order in which ECN marks are received with minimal additional complexity.

Implementation note: sending an AccECN Option each time a different counter changes and including a full-length AccECN Option on every

delayed ACK will satisfy the requirements described above and might be the easiest implementation, as long as sufficient space is available in each ACK (in total and in the option space).

Appendix A.3 gives an example algorithm to estimate the number of marked bytes from the ACE field alone, if the AccECN Option is not available.

If a host has determined that segments with the AccECN Option always seem to be discarded somewhere along the path, it is no longer obliged to follow the above rules.

3.3. AccECN Compliance by TCP Proxies, Offload Engines and other Middleboxes

A large class of middleboxes split TCP connections. Such a middlebox would be compliant with the AccECN protocol if the TCP implementation on each side complied with the present AccECN specification and each side negotiated AccECN independently of the other side.

Another large class of middleboxes intervene to some degree at the transport layer, but attempts to be transparent (invisible) to the end-to-end connection. A subset of this class of middleboxes attempts to 'normalise' the TCP wire protocol by checking that all values in header fields comply with a rather narrow interpretation of the TCP specifications. To comply with the present AccECN specification, such a middlebox MUST NOT change the ACE field or the AccECN Option and it MUST attempt to preserve the timing of each ACK (for example, if it coalesced ACKs it would not be AccECN-compliant). A middlebox claiming to be transparent at the transport layer MUST forward the AccECN TCP Option unaltered, whether or not the length value matches one of those specified in Section 3.2.3, and whether or not the initial values of the byte-counter fields are correct. This is because blocking apparently invalid values does not improve security (because AccECN hosts are required to ignore invalid values anyway), while it prevents the standardised set of values being extended in future (because outdated normalisers would block updated hosts from using the extended AccECN standard).

Hardware to offload certain TCP processing represents another large class of middleboxes, even though it is often a function of a host's network interface and rarely in its own 'box'. Leeway has been allowed in the present AccECN specification in the expectation that offload hardware could comply and still serve its function. Nonetheless, such hardware MUST attempt to preserve the timing of each ACK (for example, if it coalesced ACKs it would not be AccECN-compliant).

4. Interaction with Other TCP Variants

This section is informative, not normative.

4.1. Compatibility with SYN Cookies

A TCP server can use SYN Cookies (see Appendix A of [RFC4987]) to protect itself from SYN flooding attacks. It places minimal commonly used connection state in the SYN/ACK, and deliberately does not hold any state while waiting for the subsequent ACK (e.g. it closes the thread). Therefore it cannot record the fact that it entered AccECN mode for both half-connections. Indeed, it cannot even remember whether it negotiated the use of classic ECN [RFC3168].

Nonetheless, such a server can determine that it negotiated AccECN as follows. If a TCP server using SYN Cookies supports AccECN and if the first ACK it receives contains an ACE field with the value 0b110 or 0b111, it can assume that:

- o the TCP client must have requested AccECN support on the SYN
- o it (the server) must have confirmed that it supported AccECN

Therefore the server can switch itself into AccECN mode, and continue as if it had never forgotten that it switched itself into AccECN mode earlier.

4.2. Compatibility with Other TCP Options and Experiments

AccECN is compatible (at least on paper) with the most commonly used TCP options: MSS, time-stamp, window scaling, SACK and TCP-AO. It is also compatible with the recent promising experimental TCP options TCP Fast Open (TFO [RFC7413]) and Multipath TCP (MPTCP [RFC6824]). AccECN is friendly to all these protocols, because space for TCP options is particularly scarce on the SYN, where AccECN consumes zero additional header space.

When option space is under pressure from other options, Section 3.2.5 provides guidance on how important it is to send an AccECN Option and whether it needs to be a full-length option.

4.3. Compatibility with Feedback Integrity Mechanisms

The ECN Nonce [RFC3540] is an experimental IETF specification intended to allow a sender to test whether ECN CE markings (or losses) introduced in one network are being suppressed by the receiver or anywhere else in the feedback loop, such as another network or a middlebox. The ECN nonce has not been deployed as far

as can be ascertained. The nonce would now be nearly impossible to deploy retrospectively, because to catch a misbehaving receiver it relies on the receiver volunteering feedback information to incriminate itself. A receiver that has been modified to misbehave can simply claim that it does not support nonce feedback, which will seem unremarkable given so many other hosts do not support it either.

With minor changes AccECN could be optimised for the possibility that the ECT(1) codepoint might be used as a nonce. However, given the nonce is now probably undeployable, the AccECN design has been generalised so that it ought to be able to support other possible uses of the ECT(1) codepoint, such as a lower severity or a more instant congestion signal than CE.

Three alternative mechanisms are available to assure the integrity of ECN and/or loss signals. AccECN is compatible with any of these approaches:

- o The Data Sender can test the integrity of the receiver's ECN (or loss) feedback by occasionally setting the IP-ECN field to a value normally only set by the network (and/or deliberately leaving a sequence number gap). Then it can test whether the Data Receiver's feedback faithfully reports what it expects [I-D.moncaster-tcpm-rcv-cheat]. Unlike the ECN Nonce, this approach does not waste the ECT(1) codepoint in the IP header, it does not require standardisation and it does not rely on misbehaving receivers volunteering to reveal feedback information that allows them to be detected. However, setting the CE mark by the sender might conceal actual congestion feedback from the network and should therefore only be done sparsely.
- o Networks generate congestion signals when they are becoming congested, so they are more likely than Data Senders to be concerned about the integrity of the receiver's feedback of these signals. A network can enforce a congestion response to its ECN markings (or packet losses) using congestion exposure (ConEx) audit [I-D.ietf-conex-abstract-mech]. Whether the receiver or a downstream network is suppressing congestion feedback or the sender is unresponsive to the feedback, or both, ConEx audit can neutralise any advantage that any of these three parties would otherwise gain.

ConEx is a change to the Data Sender that is most useful when combined with AccECN. Without AccECN, the ConEx behaviour of a Data Sender would have to be more conservative than would be necessary if it had the accurate feedback of AccECN.

- o The TCP authentication option (TCP-AO [RFC5925]) can be used to detect any tampering with AccECN feedback between the Data Receiver and the Data Sender (whether malicious or accidental). The AccECN fields are immutable end-to-end, so they are amenable to TCP-AO protection, which covers TCP options by default. However, TCP-AO is often too brittle to use on many end-to-end paths, where middleboxes can make verification fail in their attempts to improve performance or security, e.g. by resegmentation or shifting the sequence space.

5. Protocol Properties

This section is informative not normative. It describes how well the protocol satisfies the agreed requirements for a more accurate ECN feedback protocol [RFC7560].

Accuracy: From each ACK, the Data Sender can infer the number of new CE marked segments since the previous ACK. This provides better accuracy on CE feedback than classic ECN. In addition if the AccECN Option is present (not blocked by the network path) the number of bytes marked with CE, ECT(1) and ECT(0) are provided.

Overhead: The AccECN scheme is divided into two parts. The essential part reuses the 3 flags already assigned to ECN in the IP header. The supplementary part adds an additional TCP option consuming up to 11 bytes. However, no TCP option is consumed in the SYN.

Ordering: The order in which marks arrive at the Data Receiver is preserved in AccECN feedback, because the Data Receiver is expected to send an ACK immediately whenever a different mark arrives.

Timeliness: While the same ECN markings are arriving continually at the Data Receiver, it can defer ACKs as TCP does normally, but it will immediately send an ACK as soon as a different ECN marking arrives.

Timeliness vs Overhead: Change-Triggered ACKs are intended to enable latency-sensitive uses of ECN feedback by capturing the timing of transitions but not wasting resources while the state of the signalling system is stable. The receiver can control how frequently it sends the AccECN TCP Option and therefore it can control the overhead induced by AccECN.

Resilience: All information is provided based on counters. Therefore if ACKs are lost, the counters on the first ACK

following the losses allows the Data Sender to immediately recover the number of the ECN markings that it missed.

Resilience against Bias: Because feedback is based on repetition of counters, random losses do not remove any information, they only delay it. Therefore, even though some ACKs are change-triggered, random losses will not alter the proportions of the different ECN markings in the feedback.

Resilience vs Overhead: If space is limited in some segments (e.g. because more option are need on some segments, such as the SACK option after loss), the Data Receiver can send AccECN Options less frequently or truncate fields that have not changed, usually down to as little as 5 bytes. However, it has to send a full-sized AccECN Option at least three times per RTT, which the Data Sender can rely on as a regular beacon or checkpoint.

Resilience vs Timeliness and Ordering: Ordering information and the timing of transitions cannot be communicated in three cases: i) during ACK loss; ii) if something on the path strips the AccECN Option; or iii) if the Data Receiver is unable to support Change-Triggered ACKs.

Complexity: An AccECN implementation solely involves simple counter increments, some modulo arithmetic to communicate the least significant bits and allow for wrap, and some heuristics for safety against fields cycling due to prolonged periods of ACK loss. Each host needs to maintain eight additional counters. The hosts have to apply some additional tests to detect tampering by middleboxes, but in general the protocol is simple to understand, simple to implement and requires few cycles per packet to execute.

Integrity: AccECN is compatible with at least three approaches that can assure the integrity of ECN feedback. If the AccECN Option is stripped the resolution of the feedback is degraded, but the integrity of this degraded feedback can still be assured.

Backward Compatibility: If only one endpoint supports the AccECN scheme, it will fall-back to the most advanced ECN feedback scheme supported by the other end.

Backward Compatibility: If the AccECN Option is stripped by a middlebox, AccECN still provides basic congestion feedback in the ACE field. Further, AccECN can be used to detect mangling of the IP ECN field; mangling of the TCP ECN flags; blocking of ECT-marked segments; and blocking of segments carrying the AccECN Option. It can detect these conditions during TCP's 3WSH so that

it can fall back to operation without ECN and/or operation without the AccECN Option.

Forward Compatibility: The behaviour of endpoints and middleboxes is carefully defined for all reserved or currently unused codepoints in the scheme, to ensure that any blocking of anomalous values is always at least under reversible policy control.

6. IANA Considerations

This document defines a new TCP option for AccECN, assigned a value of TBD1 (decimal) from the TCP option space. This value is defined as:

Kind	Length	Meaning	Reference
TBD1	N	Accurate ECN (AccECN)	RFC XXXX

[TO BE REMOVED: This registration should take place at the following location: <http://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml#tcp-parameters-1>]

Early implementation before the IANA allocation MUST follow [RFC6994] and use experimental option 254 and magic number 0xACCE (16 bits) {ToDo register this with IANA}, then migrate to the new option after the allocation.

7. Security Considerations

If ever the supplementary part of AccECN based on the new AccECN TCP Option is unusable (due for example to middlebox interference) the essential part of AccECN's congestion feedback offers only limited resilience to long runs of ACK loss (see Section 3.2.2). These problems are unlikely to be due to malicious intervention (because if an attacker could strip a TCP option or discard a long run of ACKs it could wreak other arbitrary havoc). However, it would be of concern if AccECN's resilience could be indirectly compromised during a flooding attack. AccECN is still considered safe though, because if the option is not presented, the AccECN Data Sender is then required to switch to more conservative assumptions about wrap of congestion indication counters (see Section 3.2.2 and Appendix A.2).

Section 4.1 describes how a TCP server can negotiate AccECN and use the SYN cookie method for mitigating SYN flooding attacks.

There is concern that ECN markings could be altered or suppressed, particularly because a misbehaving Data Receiver could increase its own throughput at the expense of others. Given the experimental ECN nonce is now probably undeployable, AccECN has been generalised for other possible uses of the ECT(1) codepoint to avoid obsolescence of the codepoint even if the nonce mechanism is obsoleted. AccECN is compatible with the three other schemes known to assure the integrity of ECN feedback (see Section 4.3 for details). If the AccECN Option is stripped by an incorrectly implemented middlebox, the resolution of the feedback will be degraded, but the integrity of this degraded information can still be assured.

The AccECN protocol is not believed to introduce any new privacy concerns, because it merely counts and feeds back signals at the transport layer that had already been visible at the IP layer.

8. Acknowledgements

We want to thank Koen De Schepper, Praveen Balasubramanian and Michael Welzl for their input and discussion. The idea of using the three ECN-related TCP flags as one field for more accurate TCP-ECN feedback was first introduced in the re-ECN protocol that was the ancestor of ConEx.

Bob Briscoe was part-funded by the European Community under its Seventh Framework Programme through the Reducing Internet Transport Latency (RITE) project (ICT-317700) and through the Trilogy 2 project (ICT-317756). The views expressed here are solely those of the authors.

9. Comments Solicited

Comments and questions are encouraged and very welcome. They can be addressed to the IETF TCP maintenance and minor modifications working group mailing list <tcpm@ietf.org>, and/or to the authors.

10. References

10.1. Normative References

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

- [RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <<http://www.rfc-editor.org/info/rfc3168>>.
- [RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <<http://www.rfc-editor.org/info/rfc5681>>.
- [RFC6994] Touch, J., "Shared Use of Experimental TCP Options", RFC 6994, DOI 10.17487/RFC6994, August 2013, <<http://www.rfc-editor.org/info/rfc6994>>.

10.2. Informative References

- [I-D.bensley-tcpm-dctcp] Bensley, S., Eggert, L., Thaler, D., Balasubramanian, P., and G. Judd, "Microsoft's Datacenter TCP (DCTCP): TCP Congestion Control for Datacenters", draft-bensley-tcpm-dctcp-05 (work in progress), July 2015.
- [I-D.ietf-conex-abstract-mech] Mathis, M. and B. Briscoe, "Congestion Exposure (ConEx) Concepts, Abstract Mechanism and Requirements", draft-ietf-conex-abstract-mech-13 (work in progress), October 2014.
- [I-D.kuehlewind-tcpm-ecn-fallback] Kuehlewind, M. and B. Trammell, "A Mechanism for ECN Path Probing and Fallback", draft-kuehlewind-tcpm-ecn-fallback-01 (work in progress), September 2013.
- [I-D.moncaster-tcpm-rcv-cheat] Moncaster, T., Briscoe, B., and A. Jacquet, "A TCP Test to Allow Senders to Identify Receiver Non-Compliance", draft-moncaster-tcpm-rcv-cheat-03 (work in progress), July 2014.
- [RFC3540] Spring, N., Wetherall, D., and D. Ely, "Robust Explicit Congestion Notification (ECN) Signaling with Nonces", RFC 3540, DOI 10.17487/RFC3540, June 2003, <<http://www.rfc-editor.org/info/rfc3540>>.
- [RFC4987] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", RFC 4987, DOI 10.17487/RFC4987, August 2007, <<http://www.rfc-editor.org/info/rfc4987>>.

- [RFC5562] Kuzmanovic, A., Mondal, A., Floyd, S., and K. Ramakrishnan, "Adding Explicit Congestion Notification (ECN) Capability to TCP's SYN/ACK Packets", RFC 5562, DOI 10.17487/RFC5562, June 2009, <<http://www.rfc-editor.org/info/rfc5562>>.
- [RFC5925] Touch, J., Mankin, A., and R. Bonica, "The TCP Authentication Option", RFC 5925, DOI 10.17487/RFC5925, June 2010, <<http://www.rfc-editor.org/info/rfc5925>>.
- [RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <<http://www.rfc-editor.org/info/rfc6824>>.
- [RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<http://www.rfc-editor.org/info/rfc7413>>.
- [RFC7560] Kuehlewind, M., Ed., Scheffenegger, R., and B. Briscoe, "Problem Statement and Requirements for Increased Accuracy in Explicit Congestion Notification (ECN) Feedback", RFC 7560, DOI 10.17487/RFC7560, August 2015, <<http://www.rfc-editor.org/info/rfc7560>>.

Appendix A. Example Algorithms

This appendix is informative, not normative. It gives example algorithms that would satisfy the normative requirements of the AccECN protocol. However, implementers are free to choose other ways to implement the requirements.

A.1. Example Algorithm to Encode/Decode the AccECN Option

The example algorithms below show how a Data Receiver in AccECN mode could encode its CE byte counter `r.ceb` into the ECEB field within the AccECN TCP Option, and how a Data Sender in AccECN mode could decode the ECEB field into its byte counter `s.ceb`. The other counters for bytes marked ECT(0) and ECT(1) in the AccECN Option would be similarly encoded and decoded.

It is assumed that each local byte counter is an unsigned integer greater than 24b (probably 32b), and that the following constant has been assigned:

$$\text{DIVOPT} = 2^{24}$$

Every time a CE marked data segment arrives, the Data Receiver increments its local value of `r.ceb` by the size of the TCP Data. Whenever it sends an ACK with the AccECN Option, the value it writes into the ECEB field is

$$\text{ECEB} = \text{r.ceb} \% \text{DIVOPT}$$

where `'%'` is the modulo operator.

On the arrival of an AccECN Option, the Data Sender uses the TCP acknowledgement number and any SACK options to calculate `newlyAckedB`, the amount of new data that the ACK acknowledges in bytes. If `newlyAckedB` is negative it means that a more up to date ACK has already been processed, so this ACK has been superseded and the Data Sender has to ignore the AccECN Option. Then the Data Sender calculates the minimum difference `d.ceb` between the ECEB field and its local `s.ceb` counter, using modulo arithmetic as follows:

```
if (newlyAckedB >= 0) {  
    d.ceb = (ECEB + DIVOPT - (s.ceb % DIVOPT)) % DIVOPT  
    s.ceb += d.ceb  
}
```

For example, if `s.ceb` is 33,554,433 and ECEB is 1461 (both decimal), then


```
s.ceb % DIVOPT = 1
d.ceb = (1461 + 2^24 - 1) % 2^24
      = 1460
s.ceb = 33,554,433 + 1460
      = 33,555,893
```

A.2. Example Algorithm for Safety Against Long Sequences of ACK Loss

The example algorithms below show how a Data Receiver in AccECN mode could encode its CE packet counter `r.cep` into the ACE field, and how the Data Sender in AccECN mode could decode the ACE field into its `s.cep` counter. The Data Sender's algorithm includes code to heuristically detect a long enough unbroken string of ACK losses that could have concealed a cycle of the congestion counter in the ACE field of the next ACK to arrive.

Two variants of the algorithm are given: i) a more conservative variant for a Data Sender to use if it detects that the AccECN Option is not available (see Section 3.2.2 and Section 3.2.4); and ii) a less conservative variant that is feasible when complementary information is available from the AccECN Option.

A.2.1. Safety Algorithm without the AccECN Option

It is assumed that each local packet counter is a sufficiently sized unsigned integer (probably 32b) and that the following constant has been assigned:

$$\text{DIVACE} = 2^3$$

Every time a CE marked packet arrives, the Data Receiver increments its local value of `r.cep` by 1. It repeats the same value of ACE in every subsequent ACK until the next CE marking arrives, where

$$\text{ACE} = \text{r.cep} \% \text{DIVACE}.$$

If the Data Sender received an earlier value of the counter that had been delayed due to ACK reordering, it might incorrectly calculate that the ACE field had wrapped. Therefore, on the arrival of every ACK, the Data Sender uses the TCP acknowledgement number and any SACK options to calculate `newlyAckedB`, the amount of new data that the ACK acknowledges. If `newlyAckedB` is negative it means that a more up to date ACK has already been processed, so this ACK has been superseded and the Data Sender has to ignore the AccECN Option. If `newlyAckedB` is zero, to break the tie the Data Sender could use timestamps (if present) to work out `newlyAckedT`, the amount of new time that the ACK acknowledges. Then the Data Sender calculates the minimum difference

d.cep between the ACE field and its local s.cep counter, using modulo arithmetic as follows:

```
if ((newlyAcedB > 0) || (newlyAcedB == 0 && newlyAcedT > 0))
    d.cep = (ACE + DIVACE - (s.cep % DIVACE)) % DIVACE
```

Section 3.2.2 requires the Data Sender to assume that the ACE field did cycle if it could have cycled under prevailing conditions. The 3-bit ACE field in an arriving ACK could have cycled and become ambiguous to the Data Sender if a row of ACKs goes missing that covers a stream of data long enough to contain 8 or more CE marks. We use the word 'missing' rather than 'lost', because some or all the missing ACKs might arrive eventually, but out of order. Even if some of the lost ACKs are piggy-backed on data (i.e. not pure ACKs) retransmissions will not repair the lost AcceECN information, because AcceECN requires retransmissions to carry the latest AcceECN counters, not the original ones.

The phrase 'under prevailing conditions' allows the Data Sender to take account of the prevailing size of data segments and the prevailing CE marking rate just before the sequence of ACK losses. However, we shall start with the simplest algorithm, which assumes segments are all full-sized and ultra-conservatively it assumes that ECN marking was 100% on the forward path when ACKs on the reverse path started to all be dropped. Specifically, if newlyAcedB is the amount of data that an ACK acknowledges since the previous ACK, then the Data Sender could assume that this acknowledges newlyAcedPkt full-sized segments, where newlyAcedPkt = newlyAcedB/MSS. Then it could assume that the ACE field incremented by

```
dSafer.cep = newlyAcedPkt - ((newlyAcedPkt - d.cep) % DIVACE),
```

For example, imagine an ACK acknowledges newlyAcedPkt=9 more full-size segments than any previous ACK, and that ACE increments by a minimum of 2 CE marks (d.cep=2). The above formula works out that it would still be safe to assume 2 CE marks (because $9 - ((9-2) \% 8) = 2$). However, if ACE increases by a minimum of 2 but acknowledges 10 full-sized segments, then it would be necessary to assume that there could have been 10 CE marks (because $10 - ((10-2) \% 8) = 10$).

Implementers could build in more heuristics to estimate prevailing average segment size and prevailing ECN marking. For instance, newlyAcedPkt in the above formula could be replaced with newlyAcedPktHeur = newlyAcedPkt*p*MSS/s, where s is the prevailing segment size and p is the prevailing ECN marking probability. However, ultimately, if TCP's ECN feedback becomes inaccurate it still has loss detection to fall back on. Therefore, it would seem safe to implement a simple algorithm, rather than a perfect one.

The simple algorithm for dSafer.cep above requires no monitoring of prevailing conditions and it would still be safe if, for example, segments were on average at least 5% of full-sized as long as ECN marking was 5% or less. Assuming it was used, the Data Sender would increment its packet counter as follows:

```
s.cep += dSafer.cep
```

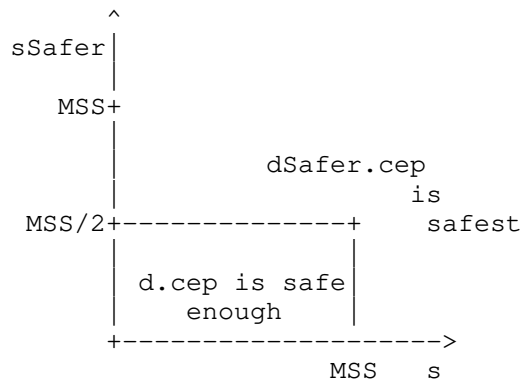
If missing acknowledgement numbers arrive later (due to reordering), Section 3.2.2 says "the Data Sender MAY attempt to neutralise the effect of any action it took based on a conservative assumption that it later found to be incorrect". To do this, the Data Sender would have to store the values of all the relevant variables whenever it made assumptions, so that it could re-evaluate them later. Given this could become complex and it is not required, we do not attempt to provide an example of how to do this.

A.2.2. Safety Algorithm with the AcceECN Option

When the AcceECN Option is available on the ACKs before and after the possible sequence of ACK losses, if the Data Sender only needs CE-marked bytes, it will have sufficient information in the AcceECN Option without needing to process the ACE field. However, if for some reason it needs CE-marked packets, if dSafer.cep is different from d.cep, it can calculate the average marked segment size that each implies to determine whether d.cep is likely to be a safe enough estimate. Specifically, it could use the following algorithm, where d.ceb is the amount of newly CE-marked bytes (see Appendix A.1):

```
SAFETY_FACTOR = 2
if (dSafer.cep > d.cep) {
    s = d.ceb/d.cep
    if (s <= MSS) {
        sSafer = d.ceb/dSafer.cep
        if (sSafer < MSS/SAFETY_FACTOR)
            dSafer.cep = d.cep      % d.cep is a safe enough estimate
    } % else
        % No need for else; dSafer.cep is already correct,
        % because d.cep must have been too small
}
```

The chart below shows when the above algorithm will consider d.cep can replace dSafer.cep as a safe enough estimate of the number of CE-marked packets:



The following examples give the reasoning behind the algorithm, assuming $MSS=1,460$ [B]:

- o if $d.cep=0$, $dSafer.cep=8$ and $d.ceb=1,460$, then $s=infinity$ and $sSafer=182.5$.
Therefore even though the average size of 8 data segments is unlikely to have been as small as $MSS/8$, $d.cep$ cannot have been correct, because it would imply an average segment size greater than the MSS .
- o if $d.cep=2$, $dSafer.cep=10$ and $d.ceb=1,460$, then $s=730$ and $sSafer=146$.
Therefore $d.cep$ is safe enough, because the average size of 10 data segments is unlikely to have been as small as $MSS/10$.
- o if $d.cep=7$, $dSafer.cep=15$ and $d.ceb=10,200$, then $s=1,457$ and $sSafer=680$.
Therefore $d.cep$ is safe enough, because the average data segment size is more likely to have been just less than one MSS , rather than below $MSS/2$.

If pure ACKs were allowed to be ECN-capable, missing ACKs would be far less likely. However, because [RFC3168] currently precludes this, the above algorithm assumes that pure ACKs are not ECN-capable.

A.3. Example Algorithm to Estimate Marked Bytes from Marked Packets

If the AccECN Option is not available, the Data Sender can only decode CE-marking from the ACE field in packets. Every time an ACK arrives, to convert this into an estimate of CE-marked bytes, it needs an average of the segment size, s_{ave} . Then it can add or subtract s_{ave} from the value of $d.ceb$ as the value of $d.cep$ increments or decrements.

To calculate `s_ave`, it could keep a record of the byte numbers of all the boundaries between packets in flight (including control packets), and recalculate `s_ave` on every ACK. However it would be simpler to merely maintain a counter `packets_in_flight` for the number of packets in flight (including control packets), which it could update once per RTT. Either way, it would estimate `s_ave` as:

$$s_ave \sim \text{flightsize} / \text{packets_in_flight},$$

where `flightsize` is the variable that TCP already maintains for the number of bytes in flight. To avoid floating point arithmetic, it could right-bit-shift by `lg(packets_in_flight)`, where `lg()` means log base 2.

An alternative would be to maintain an exponentially weighted moving average (EWMA) of the segment size:

$$s_ave = a * s + (1-a) * s_ave,$$

where `a` is the decay constant for the EWMA. However, then it is necessary to choose a good value for this constant, which ought to depend on the number of packets in flight. Also the decay constant needs to be power of two to avoid floating point arithmetic.

A.4. Example Algorithm to Beacon AccECN Options

Section 3.2.5 requires a Data Receiver to beacon a full-length AccECN Option at least 3 times per RTT. This could be implemented by maintaining a variable to store the number of ACKs (pure and data ACKs) since a full AccECN Option was last sent and another for the approximate number of ACKs sent in the last round trip time:

```
if (acks_since_full_last_sent > acks_in_round / BEACON_FREQ)
    send_full_AccECN_Option()
```

For optimised integer arithmetic, `BEACON_FREQ = 4` could be used, rather than 3, so that the division could be implemented as an integer right bit-shift by `lg(BEACON_FREQ)`.

In certain operating systems, it might be too complex to maintain `acks_in_round`. In others it might be possible by tagging each data segment in the retransmit buffer with the number of ACKs sent at the point that segment was sent. This would not work well if the Data Receiver was not sending data itself, in which case it might be necessary to beacon based on time instead, as follows:

```
if (time_now > time_last_option_sent + RTT / BEACON_FREQ)
    send_full_AccECN_Option()
```

However, this time-based approach does not work well when all the ACKs are sent early in each round trip, as is the case during slow-start.

{ToDo: A simple and robust beaconing algorithm for all circumstances is still work-in-progress.}

A.5. Example Algorithm to Count Not-ECT Bytes

A Data Sender in AccECN mode can infer the amount of TCP payload data arriving at the receiver marked Not-ECT from the difference between the amount of newly ACKed data and the sum of the bytes with the other three markings, d.ceb, d.e0b and d.elb. Note that, because r.e0b is initialised to 1 and the other two counters are initialised to 0, the initial sum will be 1, which matches the initial offset of the TCP sequence number on completion of the 3WHS.

For this approach to be precise, it has to be assumed that spurious (unnecessary) retransmissions do not lead to double counting. This assumption is currently correct, given that RFC 3168 requires that the Data Sender marks retransmitted segments as Not-ECT. However, the converse is not true; necessary transmissions will result in under-counting.

However, such precision is unlikely to be necessary. The only known use of a count of Not-ECT marked bytes is to test whether equipment on the path is clearing the ECN field (perhaps due to an out-dated attempt to clear, or bleach, what used to be the ToS field). To detect bleaching it will be sufficient to detect whether nearly all bytes arrive marked as Not-ECT. Therefore there should be no need to keep track of the details of retransmissions.

Appendix B. Alternative Design Choices (To Be Removed Before Publication)

This appendix is informative, not normative. It records alternative designs that the authors chose not to include in the normative specification, but which the IETF might wish to consider for inclusion:

Feedback all four ECN codepoints on the SYN/ACK: The last two negotiation combinations in Table 2 could also be used to indicate AccECN support and to feedback that the arriving SYN was ECT(0) or ECT(1). This could be used to probe the client to server path for incorrect forwarding of the ECN field [I-D.kuehlewind-tcpm-ecn-fallback]. Note, however, that it would be unremarkable if ECN on the SYN was zeroed by security devices,

given RFC 3168 prohibited ECT on SYN because it enables DoS attacks.

Feedback all four ECN codepoints on the First ACK: To probe the server to client path for incorrect ECN forwarding, it could be useful to have four feedback states on the first ACK from the TCP client. This could be achieved by assigning four combinations of the ECN flags in the main TCP header, and only initialising the ACE field on subsequent segments.

Empty AccECN Option: It might be useful to allow an empty (Length=2) AccECN Option on the SYN/ACK and first ACK. Then if a host had to omit the option because there was insufficient space for a larger option, it would not give the impression to the other end that a middlebox had stripped the option.

Appendix C. Open Protocol Design Issues (To Be Removed Before Publication)

1. Currently it is specified that the receiver 'SHOULD' use Change-Triggered ACKs. It is controversial whether this ought to be a 'MUST' instead. A 'SHOULD' would leave the Data Sender uncertain whether it can rely on the timing and ordering information in ACKs. If the sender guesses wrongly, it will probably introduce at least 1RTT of delay before it can use this timing information. Ironically it will most likely be wanting this information to reduce ramp-up delay. A 'MUST' could make it hard to implement AccECN in offload hardware. However, it is not known whether AccECN would be hard to implement in such hardware even with a 'SHOULD' here. For instance, was it hard to offload DCTCP to hardware because of change-triggered ACKs, or was this just one of many reasons? The choice between MUST and SHOULD here is critical. Before that choice is made, a clear use-case for certainty of timing and ordering information is needed, plus well-informed discussion about hardware offload constraints.
2. There is possibly a concern that a receiver could deliberately omit the AccECN Option pretending that it had been stripped by a middlebox. No known way can yet be contrived to take advantage of this downgrade attack, but it is mentioned here in case someone else can contrive one.
3. The s.cep counter might increase even if the s.ceb counter does not (e.g. due to a CE-marked control packet). The sender's response to such a situation is considered out of scope, because this ought to be dealt with in whatever future specification allows ECN-capable control packets. However, it is possible that the situation might arise even if the sender has not sent ECN-

capable control packets, in which case, this draft might need to give some advice on how the sender should respond.

Appendix D. Changes in This Version (To Be Removed Before Publication)

The difference between any pair of versions can be displayed at
<<http://datatracker.ietf.org/doc/draft-kuehlewind-tcpm-accurate-ecn/history/>>

From 04 to 05::

- * Corrected ambiguity between Classic ECN and Classic ECN feedback throughout
- * Changed MUST to SHOULD send AccECN option on SYN/ACK last ACK of 3WHS and first data segment from client, to allow for cached knowledge of option traversal problems.
- * Removed duplication of normative language about sending a full-length option in the sections on "The AccECN Option" and "Usage of the AccECN Option", and mutually cross referenced.
- * Acknowledged Koen De Schepper and Praveen Balasubramanian
- * Noted in Appendix that algo to beacon a full-length option is work-in-progress
- * Editorial corrections and clarifications throughout

Authors' Addresses

Bob Briscoe
Simula Research Laboratory

EMail: ietf@bobbriscoe.net
URI: <http://bobbriscoe.net/>

Mirja Kuehlewind
ETH Zurich
Gloriastrasse 35
Zurich 8092
Switzerland

EMail: mirja.kuehlewind@tik.ee.ethz.ch

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

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

tcpm
Internet-Draft
Intended status: Experimental
Expires: January 14, 2013

M. Kuehlewind, Ed.
University of Stuttgart
R. Scheffenegger
NetApp, Inc.
July 13, 2012

Accurate ECN Feedback Option in TCP
draft-kuehlewind-tcpm-accurate-ecn-option-01

Abstract

This document specifies an TCP option to get accurate Explicit Congestion Notification (ECN) feedback from the receiver. 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 feedback information in the case where more than one marking is received in one RTT. This TCP extension can be used in addition to the classic ECN as well as with a more accurate ECN scheme recently proposed which reuses the ECN bit in the TCP header.

Status of this Memo

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

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

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

This Internet-Draft will expire on January 14, 2013.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal

Provisions Relating to IETF Documents
(<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Overview ECN and ECN Nonce in IP	3
1.2. Requirements Language	3
2. Negotiation of Accurate ECN feedback	4
3. Accurate ECN (AccECN) feedback Option Specification	5
4. Acknowledgements	6
5. IANA Considerations	6
6. Security Considerations	6
7. References	6
7.1. Normative References	6
7.2. Informative References	6
Authors' Addresses	7

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 feedback information in case when more than one marking is received in one RTT.

This documents specifies an TCP option to provide more than one ECN feedback signal per RTT. This modification does not obsolete [RFC3168]. This TCP extension can be used in addition to the classic ECN as well as in addition to more accurate ECN scheme recently proposed which reuses the ECN bits in the TCP header for the same purpose than this extension --- more accurate ECN feedback (see [I-D.kuehlewind-conex-accurate-ecn]). Note that a new TCP extension can experience deployment problems by middleboxes dropping unknown options. Thus the ECN feedback in the TCP header is still needed to ensure ECN feedback. Moreover, this option will increase the header length for all kind of TCP packets which can cause additional load in case of severe congestion (on the feedback channel).

1.1. Overview ECN and ECN Nonce in IP

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 (ETC) which results in two signals --- ECT(0) and respectively ECT(1). A network node can set both bits simultaneously when it experiences congestion. When both bits are set the packets is regarded as "Congestion Experienced" (CE).

ECN-Nonce [RFC3540] is an optional addition to ECN that is used to protects the TCP sender against accidental or malicious concealment of marked or dropped packets. With ECN-Nonce a nonce sum is maintain that counts the occurrence of ECT(1) packets.

1.2. 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)/ECT(1): either one of the two ECN-Capable Transport codepoints.

In this document, we will call the ECN feedback scheme as specified in [RFC3168] the 'classic ECN'. A 'congestion mark' is defined as an IP packet where the CE codepoint is set.

2. Negotiation of Accurate ECN feedback

As there is only limited space in the TCP Options, particularly during the initial three-way handshake, an abbreviated Option is used to negotiate for Accurate ECN feedback. This option also initiates all counters to an initial value of zero at the receiving side.

TCP Accurate ECN Option Negotiation:

Kind: TBD

Length: 2 bytes

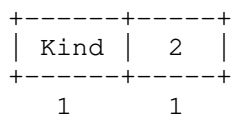


Figure 1: Accurate ECN feedback TCP option negotiation

This abbreviated option is only valid in a <SYN> or <SYN,ACK> segment, during a three way handshake. The negotiation follows the same procedure as with other TCP options, i.e. SACK. A TCP sender MAY send the accurate ECN feedback negotiation option in an initial SYN segment and MAY send a more accurate ECN option (see Section 3) in other segments only if it received this option negotiation in the initial <SYN> segment or <SYN,ACK> for the connection. A TCP receiver MAY send an <SYN,ACK> segment with the accurate ECN feedback negotiation option in response to a received accurate ECN feedback negotiation option in the <SYN>. If both ends indicate that they support Accurate ECN (AcceCN) feedback, the AcceCN option SHOULD be used in any subsequent TCP segment. A TCP sender or receiver MUST only negotiate for the AcceCN option if ECN is negotiated as well.

3. Accurate ECN (AccECN) feedback Option Specification

A TCP receiver, that provides Accurate ECN feedback, will maintain a counter for the number of ECT(0), ECT(1), CE, non-ECT marked and lost packets as well as the cumulative number of bytes of CE marked packets. The TCP option to provide the Accurate ECN (AccECN) feedback to the sender will echo these counters.

TCP Accurate ECN Option:

Kind: TBD (same as above)

Length: 12 bytes

Kind	12	ECT(0)	ECT(1)	CE	non-ECT	loss	CE in bytes
1	1	2	2	1	1	1	3

Figure 2: Accurate ECN feedback TCP option

TCP anyway provides a mechanism to detect loss as loss should always be assumed as a strong signal for congestion and TCP congestion control reacts on loss. If TCP SACK is not available, the exact number of losses is not known. Moreover, the TCP loss detection (incl. SACK) is done in bytes and not in number of packets. The number of lost packets can be used by the sender to calculate the ECN Nonce sum more exactly.

The same feedback information are proposed for the (ECN) feedback in RTP (see [I-D.ietf-avtcore-ecn-for-rtsp]).

As TCP is a bi-directional protocol, this option can be used in both directions. With the reception of every data segment at least one of the counters changes (ECT(0) or ECT(1)). The AccECN option SHOULD be included in every ACK to ensure the reception of the ECN feedback at the sender in case of ACK loss. To reduce network load the AccECN option MAY not be sent in every ACK, e.g. only in very second ACK (if ACKs are sent very frequently).

In general it is possible that any of the counters wraps around. In this case the information might get corrupted if e.g. for any reason only one ACK per RTT is sent and more than 256 CE marks occur in one RTT. For this case it MUST be ensured, that at least three ACKs/segments with the AccECN option have been sent prior to the counter experiencing an wrap around. Whenever an AccECN Option is received with smaller counter value than in the previous one and the

respective ACK acknowledges new data, a wrap around MUST be assumed.

4. Acknowledgements

5. IANA Considerations

TBD

6. Security Considerations

TBD

7. References

7.1. 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.

7.2. 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.ietf-avtcore-ecn-for-rtp] Westerlund, M., Johansson, I., Perkins, C., O'Hanlon, P., and K. Carlberg, "Explicit Congestion Notification (ECN) for RTP over UDP", draft-ietf-avtcore-ecn-for-rtp-08 (work

in progress), May 2012.

[I-D.kuehlewind-conex-accurate-ecn]

Kuehlewind, M. and R. Scheffenegger, "Accurate ECN
Feedback in TCP", draft-kuehlewind-conex-accurate-ecn-01
(work in progress), October 2011.

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

