TCP Maintenance and Minor Extensions (tcpm)
Internet-Draft
Intended status: Standards Track
Expires: January 21, 2016

A. Zimmermann
R. Scheffenegger
NetApp, Inc.

July 20, 2015

Using the TCP Echo Option for Spurious Retransmission Detection draft-zimmermann-tcpm-spurious-rxmit-00

Abstract

The Spurious Retransmission Detection (SRD) algorithm allows a TCP sender to always detect if it has entered loss recovery unnecessarily. It requires that both the TCP Echo option defined in [I-D.zimmermann-tcpm-echo-option], and the SACK option [RFC2018] be enabled for a connection. The SRD algorithm makes use of the fact that the TCP Echo option, used in conjunction with the SACK feedback, can be used to completely eliminate the retransmission ambiguity in TCP. Based on the reflected data contained in the first acceptable ACK that arrives during loss recovery, it decides whether loss recovery was entered unnecessarily. The SRD mechanism further enables improvements in loss recovery. This includes a TCP enhancement to detect and quickly resend lost retransmissions.

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 21, 2016.

Copyright Notice

Table of Contents

1. Introduction

Using only the sequence number, a TCP sender is not able to distinguish whether the first ACK, acknowledging new data, that arrives after a retransmit, was sent in response to the original transmit or the retransmission. This effect is known as the retransmission ambiguity problem [Zh86], [KP87]. Spurious retransmissions, where a segment is sent multiple times, can be caused by packet reordering, packet duplication, or a sudden delay increase in the data or the ACK path. All these cases are preceded by either a fast retransmit or a timeout-based retransmit.

The Eifel Detection Algorithm [RFC3522] aims to address these occurrences, but falls short to completely solve the ambiguity problem due to limitations in how the TCP Timestamps option is processed by the receiver.

The TCP Timestamps option already provides a means of marking retransmitted segments differently. However, the method used by a TCP receiver when a Timestamp option is reflected precludes the use of this option in most cases. The notable exception is the recovery of lost segments, when none of the retransmissions is lost or reordered in turn. Similarly, spurious retransmissions can also only

be detected and recovered from, when all of the retransmitted packets are delivered in-order and without leaving any gaps in the receive-buffer. Elsewise, the Timestamp option does not allow a solid discrimination between original or retransmitted segments, that triggered subsequent duplicate ACKs.

The semantics of the TCP Echo option, and their treatment by a receiver are different from those of the TCP Timestamps option. That allows a complete solution to disambiguate between all retransmissions, including multiple retransmissions of the same segment, packet duplication, and reordering events.

Enhancements in the area of TCP loss recovery and spurious retransmission detection are allowed by using synergistic signaling between the TCP Echo option and the selective acknowledgment (SACK) option. This allows to completely address any retransmission ambiguity.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119]. These words only have such normative significance when in ALL CAPS, not when in lower case.

Acceptable ACK: is an ACK that acknowledges previously unacknowledged data. See [RFC0793].

Forward Acknowledgement (FACK): is the the highest sequence number known to have reached the receiver, plus one, using SACK information. See [MM96].

Lost Retransmission Detection (LRD): is a mechanism to timely detect lost retransmissions during loss recovery, and quickly send the lost segment anew instead of waiting for a retransmission timeout. A simple and limited variant, that is not formally specified, is currently in use by the Linux TCP stack.

Recover: When in fast recovery, this variable records the send sequence number that must be acknowledged before the fast recovery procedure is declared to be over. See [RFC6582].

3. The Spurious Retransmission Detection Algorithm

3.1. Motivation

In order to detect spurious retransmissions, the sender requires information to uniquely identify each retransmission of every segment sent. TCP Eifel [RFC3522] uses additional information from the TCP Timestamps option [RFC7323] for this purpose. This can remove some ambiguity, but only under limited circumstances - it only works in the absence of additional impediments like ACK reordering or multiple loss.

However, the semantics used by the receiver when reflecting back a received timestamp is such that this approach only works for the first retransmission in a window, every subsequent retransmission cannot be disambiguated from a received original transmission using timestamps in most cases.

When a segment is retransmitted without the timestamp clock increasing, Eifel detection also has no signal to differentiate if a spurious retransmission had occurred. This is of particular concern at high data rates and when the RTT is low.

Retransmission ambiguity detection during loss recovery (as opposed to the first retransmission in a window) allows an additional level of loss recovery control without reverting to timer-based methods. As with the deployment of SACK, separating "what" to send from "when" to send it, is driven one step further. In particular, less conservative loss recovery schemes, which do not trade the principle of packet conservation against timeliness, require a reliable way of prompt and best possible feedback from the receiver about any delivered segment and the ordering in which they got delivered.

SACK signaling [RFC2018] goes quite a long way, but does not suffice in all circumstances, e.g. when retransmissions are lost. Further, DSACK [RFC2883] does indicate if spurious retransmissions occured, but that signal is delayed by one RTT [RFC3708]. However, loss recovery is likely to have ended at that time. Furthermore, the DSACK option by itself will not yield the information, if the late arrived segment was the original or retransmitted segment.

Using the facility provided by the TCP Echo option a TCP sender is able to differentiate between original and retransmitted segments, even within the same TCP Timestamps options clock tick (i.e. when RTT is shorter than the TCP timestamp clock interval). In addition, as the TCP Echo option is reflected back with the most recently observed value by the receiver, all instances where Eifel detection [RFC3522] is not able to detect reliably can be addressed. Furthermore, as the sender is immediately notified which segment triggered the ACK, no delay is induced when deducting if a retransmission was spurious.

3.2. Basic Idea

Using the TCP Echo option, which has different semantics from the TCP Timestamps option, it is possible to uniquely identify and disambiguate each segment, including every retransmission. However, the value carried with the TCP Echo option does not need to be unique by itself (e.g. every segment having a different TCP Echo option value), as other information contained in the TCP Header and TCP options, namely the acknowledgment number and the SACK blocks, differentiate already between segments in the TCP stream space. Thus, it is only necessary to differentiate between segments (of the same size) covering the same sequence space.

One simple approach would be to have a per-segment counter, which is set to zero for each new transmission, and incremented whenever that same segment is retransmitted anew. However, this approach would require per-segment state in the sender. To reduce the complexity in the sender, and not require per-segment state, a simpler approach is to use a single global counter, that is increased whenever a segment has to be resent. In ECN environments, an increase of the retransmission counter is expected to typically coincide with CWR-marked segments.

Apart from simplifying the design, this also yields additional benefits when the reorder delay is larger than one RTT, and when Acknowledgments are lost or reordered. Note that the wire representation of this counter SHOULD NOT be as simplistic as described here (see Section 6).

The retransmission counter has to be large enough to cater for all expected RTOs before a TCP sender gives up and terminates a connection (see [RFC1122], section 4.2.3.5, variable R2), plus all the fast retransmissions of that segment that may have happened before triggering the chain of exponential back-off RTOs. In general, a single octet is enough to convey the retransmission counter.

The sender has to transmit every segment with a TCP Echo option. Sending the Echo option only with retransmission has the issue of adding option space, thereby potentially requiring the sender to segment the TCP payload differently (and sending an additional segment) than the original segment. A sender SHOULD therefore add the echo option to every sent segment to simplify the implementation. Sending the TCP Echo option with every segment has the added benefit to make the mechanism tolerate ACK losses.

3.3. The Algorithm

Spurious Retransmission Detection (SRD) utilizes the TCP Echo option [I-D.zimmermann-tcpm-echo-option], which is used with at least one octet of payload. If another algorithm deployed on the sender also uses the TCP Echo option on a TCP connection, it is up to the implementer to combine the necessary signaling of these mechanisms to fit into a single TCP Echo option (e.g. by mapping the Echo option codepoints into a translation table, or extending the length of the TCP Echo option and matching parts of the data to the different mechanisms).

The TCP sender maintains a single, connection-global counter. This retransmission counter MUST be increased by one whenever the sender enters loss recovery, experiences a Retransmission Timeout (RTO), or re-sends a previously already retransmitted segment once more. Care must be taken to limit a malicious receivers ability make genuine retransmissions appear as spurious retransmissions to the sender (see Section 6), when encoding the internal counter value to the wire representation.

Every transmitted segment carries a TCP Echo option, where the data reflects the current value of the sender’s retransmission counter. When the sender receives an ACK, the TCP Echo option data is extracted and checked against the current value of the retransmission counter, together with a check if the ACK is acceptable. Note that information from not acceptable ACKs MUST be evaluated too.

After a retransmission has been sent, either due to a Fast Retransmission or an RTO, the first acceptable ACK is checked. If the received retransmission counter is equal to the current counter value maintained by the sender, a valid retransmission was sent. If the received value is less than the current retransmission counter, a spurious retransmission was sent, and if no valid retransmissions are detected until the end of the loss recovery phase, the TCP sender MAY restore the congestion control state to the state prior to entering loss recovery. Even if some of the retransmissions of this loss recovery phase may have been spurious, the TCP sender MUST NOT react by restoring the congestion control state to the state before entering loss recovery, if any of the retransmissions are deduced to be valid.

A TCP sender MAY retain the congestion control state for up to two RTTs since entering the loss recovery state. {TODO: Not after exiting loss recovery?} If all retransmissions that were performed in this period are later found to have been spurious - either by evaluating the retransmission counter values of received unacceptable (first duplicate) ACKs, or a DSACK [RFC3708] indication - the TCP sender MAY

revert to the stored congestion control state, e.g. by following the Eifel Response algorithm [RFC4015].

4. Examples

This section shows a few examples, from simple to increasingly complex. Some of these scenarios are addressed by exising mechanisms like Eifel, and DSACK; in particular, corner cases that are not adressed with existing mechanisms are demonstrated.

In the following examples, each set of three lines starting with "ack#", "sack:", and "sent:" represent one RTT. It is assumed that the sender has sent segments 1 to 8 in the prior RTT, and for readability, the numbers show represent full segments rather than sequence numbers.

The two lines following ("ack#" and "sack:") indicate what ACK is being triggered on the receiver. The ACK number is the sequence number of the next expected segment, followed by a dot and the value of the received TCP Echo option value - again for simpilicty, the internal representation of the global retransmission counter value (initially set to zero) is shown, not the wire representation.

In the line "sack:" the relevant SACK blocks are depicted, again with a single number representative of an entire segment. When these ACKs are seen by the sender, it will start sending the segment depicted in the line "sent:", again together with the retransmission counter value.

Further assumptions in these examples are that the sender is using proportional rate reduction [RFC6937], limited transmit [RFC3042], and selective acknowledgments (SACK) [RFC2018] and [RFC2883], is not application limited when sending data and has a congestion window of 9 segments.

1. Fast Retransmission

```
ack#   X 1.0  1.0  1.0  1.0  1.0  1.0  1.0
sack:    2    2-3  2-4  2-5  2-6  2-7  2-8
sent:    9.0  10.0 1.1       11.1      12.1

ack#     1.0  1.0  11.1      12.1      13.1
sack:    2-9  2-10
```

detected as valid retransmission, as for the first acceptable ACK (11.1) after the retransmission the Echo Tag is equal to the retransmission counter.

2. Multiple loss

```
ack#   X 1.0    1.0      1.0  1.0  1.0  1.0    X
sack:    2      2-3      2-4  2-5  2-6  2-7
sent:    9.0    10.0     1.1       11.1

ack#     1.0    1.0      8.1       8.1
sack:    2-7,9  2-7,9-10 9-10      9-11
sent:    12.1             8.1       ...
```

SRD detectes this as valid retransmission, as for the first acceptable ACK (8.1) and every other retransmission after the first retransmission the Echo Tag is equal to the retransmission counter. Retransmission counter is not increased when sending (8.1) as loss recovery was not yet exited at the time of sending that retransmission.

3. Retransmission Timeout (RTO)

```
ack#  X   X    X    X    X    X    X    X
sack:
sent:    ----- RTO -->

ack#
sack:
sent:    ----- RTO --> 1.1

ack#                   1.1
sack:
```

detected as valid retransmission, as the first acceptable ACK (1.1) after the retransmission contains the Echo Tag of the retransmission.

4. Retransmission loss

```
ack#   X 1.0  1.0  1.0  1.0  1.0  1.0  1.0
sack:    2    2-3  2-4  2-5  2-6  2-7  2-8
sent:    9.0  10.0 1.1       11.1      12.1
                   X
ack#     1.0  1.0            1.1       1.1
sack:    2-9  2-10           2-11      2-12
```

no acceptable ack, but a jump on the counter tag to the current counter. (see {TODO: LRD document}), also FACK is larger than Recovery Point (The condition of FACK > RP will trigger linux LRD).

Note: without LRD, the lost retransmission will NOT be retried before an RTO. Can not be detected by Eifel due to TCP Timestamps semantics.

5. Multiple loss, first retransmission lost

```
ack#   X X    1.0  1.0  1.0  1.0  1.0  1.0
sack:         3    3-4  3-5  3-6  3-7  3-8
sent:         9.0  1.1       2.1       10.1
                   X
ack#          1.0            1.1       1.1
sack:         3-9            2-9       2-10
sent:         11.1           1.2       12.2
```

no acceptable ack, but a jump on the counter tag to the current counter. see {TODO: LRD document}. Linux LRD would delay the sending of 1.2 until after FACK passes RP (in this example, the last two sent segments was be swapped). Not detectable by Eifel.

6. RTT > Reordering delay > DupThresh

```
                                r
ack#   R 1.0  1.0  1.0  1.0  6.0  7.0  8.0
sack:    2    2-3  2-4  2-5
sent:    8.0  9.0  1.1       10.1 11.1 12.1

ack#     9.0  10.0 10.1      11.1 12.1 13.1
sack:              1
```

detected as spurious retransmission, as the first acceptable ACK (6.0) after the retransmission is received with the Echo Tag unequal the current retransmission counter; DSACK detects this 1 RTT later; Eifel detects this at the same time using timestamps

7. Reordering delay > RTT

```
ack#   R 1.0  1.0  1.0  1.0  1.0  1.0  1.0
sack:    2    2-3  2-4  2-5  2-6  2-7  2-8
sent:    9.0  10.0 1.1       11.1      12.1
                                  r
ack#     1.0  1.0  11.1      12.1 12.0 13.1
sack:    2-9  2-10               1
```

detected as valid retransmission, as the first acceptable ACK (11.1) after the retransmission contains the Echo Tag of the retransmission.

Note that at (12.0), with the retransmission counter always counting up, this detection becomes possible, by seeing 2nd ACK with lower retransmission counter (SRD) one RTT later: DSACK and SRD both detect at the same time

8. Packet duplication

SACK is mandatory for SRD, and SACK detects this as duplication event, with no further action

9. Reordering and loss

```
                                  r
ack#   R X    1.0  1.0  1.0  2.0  2.0  2.0
sack:         3    3-4  3-5  3-5  3-6  3-7
sent:         8.0  9.0  1.1       2.1

ack#:         2.0  2.0  2.1       10.1
sack          3-8  3-9  1,3-9
```

detected as spurious retransmission, as the first acceptable ACK (2.0) after the retransmission is received with the Echo Tag unequal the current retransmission counter; no undo at that point, since still in recovery. DSACK detects this 1 RTT later; Eifel detects this at the same time using timestamps.
Detected as valid retransmission, as for the second acceptable ACK (10.1) after the retransmission the Echo Tag is equal to the retransmission counter, prior to leaving loss recovery

10. Loss and reordering (reordered retransmission)

```
ack#   X 1.0  1.0  1.0  1.0  1.0  1.0  1.0
sack:    2    2-3  2-4  2-5  2-6  2-7  2-8
sent:    9.0  10.0 1.1       11.1      12.1
                  R
                                    r
ack#     1.0  1.0            1.1  12.1 13.1
sack:    2-9  2-10           2-11
sent:         13.1           1.2  14.2 15.2

ack#          14.1           14.2 15.2 16.2
sack:                        1
```

reordered retransmission

LRD triggered (no acceptable ack, when retransmission count increases - {TODO: LRD document}), also FACK > Recovery Point (Linux LRD) Detected as spurious retransmission, as the first acceptable ACK (12.1) after the 2nd retransmission is received with the Echo Tag unequal the current retransmission counter; undo at that point, since recovery is exited at the same time. DSACK detects this 1 RTT later; Eifel detects this at the same time using timestamps.

11. ACK reordering after loss

```
ack#   X 1.0  1.0  1.0  1.0  1.0  1.0  1.0
sack:    2    2-3  2-4  2-5  2-6  2-7  2-8
sent:    9.0  10.0 1.1       11.1      12.1
                   R
                                   r
ack#     1.0  1.0            1.1  11.1 13.1
sack:    2-9  2-10           2-11
sent:         13.1           1.2  14.2 15.2
```

valid retransmission, as first acceptable ack (11.1) after retransmission has same retransmission counter as the current value. Reordered ACK has still same (not lower!) retransmission counter.

12. ACK reordering after reordering

```
                                   rR
ack#   R 1.0  1.0  1.0  1.0  7.0  6.0  8.0
sack:    2    2-3  2-4  2-5
sent:    8.0  9.0  1.1       10.1      11.1

ack#     9.0  10.0 10.1      11.1      12.1
sack:              1
```

detected as spurious retransmission, as the first acceptable ACK (7.0) after the retransmission is received with the Echo Tag unequal the current retransmission counter; DSACK detects this 1 RTT later; Eifel detects this at the same time using timestamps

13. ACK loss after reordering

```
                                  r
ack#   R 1.0  1.0  1.0  1.0  (6.0) 7.0  8.0
sack:    2    2-3  2-4  2-5
sent:    8.0  9.0  1.1             10.1 11.1

ack#     9.0  10.0 10.1            11.1 12.1
sack:              1
```

detected as spurious retransmission, as the first acceptable ACK (7.0) after the retransmission is received with the Echo Tag unequal the current retransmission counter; DSACK detects this 1 RTT later; Eifel detects this at the same time using timestamps
Note that retransmission counter only increasing helps this case to work both with reordering (spurious retransmission) and retransmission ACK loss - the relevant information is conveyed for about 1RTT thus single ACK loss does not impact the detection.

14. TODO: delay ACK

Todo: Example necessary?

5. IANA Considerations

This document contains no requests to IANA, as only a new combined use of TCP options is described.

6. Security Considerations

This document describes a new use for the TCP Echo option. Transporting the retransmission counter in the clear may pose a security problem when the TCP sender uses SRD to restore the TCP state. A malicious receiver could game the sender to always restore the congestion control state to the one preceding the lost recovery episode, thereby making the sender not back off its transmission rate.

As the sender can put any data into the TCP Echo option, the transmission counter value can be masked in various ways. A TCP sender can map the same counter value to multiple TCP Echo option data values, and track which of these data values would be expected for a given acknowledgement. Alternatively, the TCP Echo option data could be a (secure) hash of the sequence number of the sent segment, a random, per-connection secret, and the retransmission counter. The TCP Echo data would look rather as random sequence of octets in both cases, making it very hard for a malicious receiver to obtain an unfair share of bandwidth by masking genuine retransmissions as spurious.

7. Acknowledgements

The authors like to thank Bob Briscoe and Brian Trammel for their invaluable input.

Alexander Zimmermann the European Union's Horizon 2020 research and innovation program 2014-2018 under grant agreement No. 644866 (SSICLOPS). This document reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

8. References

8.1. Normative References

[I-D.zimmermann-tcpm-echo-option] Zimmermann, A., Scheffenegger, R., and B. Briscoe, "The TCP Echo and TCP Echo Reply Options", draft-zimmermann-tcpm-echo-option-00 (work in progress), June 2015.

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

8.2. Informative References

[KP87] Karn, P. and C. Partridge, "Estimating Round-Trip Times in Reliable Transport Protocols", Proc. SIGCOMM '87, August 1987.

[MM96] Mathis, M. and J. Mahdavi, "Forward Acknowledgement: Refining TCP Congestion Control", ACM SIGCOMM 1996 Proceedings, in ACM Computer Communication Review 26 (4), pp. 281-292, October 1996.

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

[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.

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

[RFC3042] Allman, M., Balakrishnan, H., and S. Floyd, "Enhancing TCP's Loss Recovery Using Limited Transmit", RFC 3042, January 2001.

[RFC3522] Ludwig, R. and M. Meyer, "The Eifel Detection Algorithm for TCP", RFC 3522, April 2003.

[RFC3708] Blanton, E. and M. Allman, "Using TCP Duplicate Selective Acknowledgement (DSACKs) and Stream Control Transmission Protocol (SCTP) Duplicate Transmission Sequence Numbers (TSNs) to Detect Spurious Retransmissions", RFC 3708, February 2004.

[RFC4015] Ludwig, R. and A. Gurtov, "The Eifel Response Algorithm for TCP", RFC 4015, February 2005.

[RFC6582] Henderson, T., Floyd, S., Gurtov, A., and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 6582, April 2012.

[RFC6937] Mathis, M., Dukkipati, N., and Y. Cheng, "Proportional Rate Reduction for TCP", RFC 6937, May 2013.

[RFC7323] Borman, D., Braden, B., Jacobson, V., and R. Scheffenegger, "TCP Extensions for High Performance", RFC 7323, September 2014.

[Zh86] Zhang, L., "Why TCP timers don't work well", Proc. SIGCOMM '86, Sep 1986.

Authors' Addresses

Alexander Zimmermann
NetApp, Inc.
Sonnenallee 1
Kirchheim 85551
Germany

Phone: +49 89 900594712
Email: alexander.zimmermann@netapp.com

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

Email: rs@netapp.com