TCP Maintenance and Minor Extensions   F. Gont
(tcpm)   UTN-FRH / SI6 Networks
Internet-Draft   D. Borman
Updates: 793 (if approved)   Quantum Corporation
Intended status: Standards Track   March 5, 2018
Expires: September 6, 2018

On the Validation of TCP Sequence Numbers
draft-gont-tcpm-tcp-seq-validation-03.txt

Abstract

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

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

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

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

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

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. TCP Sequence Number Validation

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

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

   0       0     SEG.SEQ = RCV.NXT

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

  >0       0     not acceptable

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

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

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

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

3. Scenarios in which Undesirable Behaviors Might Arise

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

3.1. TCP simultaneous open

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

```
       TCP A                                                   TCP B

   1. CLOSED                                                  CLOSED

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

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

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

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

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

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

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

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

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

(Failed) Simultaneous Connection Synchronization

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

3.2. TCP self connects

Some systems have been found to be unable to process TCP connection requests in which the source endpoint {Source Address, Source Port} is the same as the destination end-point {Destination Address, Destination Port}. Such a scenario might arise e.g. if a process creates a socket, bind()s a local end-point (IP address and TCP port), and then issues a connect() to the same end-point as that specified to bind().

> While not widely employed in existing applications, such a socket could be employed as a "full-duplex pipe" for Inter-Process Communication (IPC).
>
> This scenario is described in detail in pp. 960-962 of [Wright1994].

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

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

3.3. TCP simultaneous close

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

```
       TCP A                                                     TCP B

    1. ESTABLISHED                                               ESTABLISHED

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

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

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

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

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

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

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

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

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

(Failed) Simultaneous Connection Termination

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

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

In line 4, the FIN segment sent by TCP A arrives to TCP B, causing it to transition to the CLOSING state (at this point, TCP B’s RCV.NXT becomes 101). In line 5, TCP A acknowledges the receipt of the TCP B’s FIN segment, and also sets the FIN bit in the outgoing segment (since it has not yet been acknowledged). In line 6, TCP B acknowledges the receipt of TCP A’s FIN segment, and also sets the FIN bit in the outgoing segment (since it has not yet been acknowledged). In line 7, the FIN/ACK from line 5 arrives at TCP B. In line 8, the FIN/ACK from line 6 fails the TCP sequence number validation check, and thus elicits a ACK segment (the segment also contains the FIN bit set, since it had not yet been acknowledged). In line 9, the FIN/ACK from line 7 fails the TCP sequence number validation check, and hence elicits an ACK segment (the segment also contains the FIN bit set, since it had not yet been acknowledged). In line 10, the FIN/ACK from line 8 finally arrives at TCP B.

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

3.4. Simultaneous Window Probes

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

```
   TCP A                                                      TCP B

1. ESTABLISHED                                                ESTABLISHED

2.                      (both TCP windows open)

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

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

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

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

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

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

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

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

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

(Failed) Simultaneous Connection Termination

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

TCP B sends a "Window Probe".

Both Window Probes cross each other in the network.

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

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

4. Updating RFC 793

4.1. TCP sequence number validation

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

---------------- cut here -------------- cut here ----------------

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

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

or

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

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

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

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

   0       0     SEG.SEQ = RCV.NXT

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

  >0       0     not acceptable

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

---------------- cut here -------------- cut here ----------------

is replaced with:

```
 ---------------- cut here -------------- cut here ----------------
A segment is judged to occupy a portion of valid receive sequence
space if

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

or

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


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

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

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

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

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

    >0       0     not acceptable

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

 ---------------- cut here -------------- cut here ----------------
```

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

---------------- cut here -------------- cut here ----------------

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

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

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

   0       0     SEG.SEQ = RCV.NXT

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

  >0       0     not acceptable

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

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

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

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

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

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

---------------- cut here -------------- cut here ----------------

is replaced with:

---------------- cut here -------------- cut here ----------------

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

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

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

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

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

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

  >0       0     not acceptable

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

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

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

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

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

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

```
       one byte to the left of the receive window.
  ---------------- cut here -------------- cut here ----------------
```

4.2. Alternative fix for TCP sequene number validation

The Linux kernel performs a slightly different TCP sequence number validation check, in that, during window probes, can acommodate window probes of any size (as opposed to the defacto standard 1-byte window probes). This makes the code more general, at the expense of additional state in the TCB (e.g., the TCP sequence number employed in the last window probe).

4.3. TCP self connects

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

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

5. IANA Considerations

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

6. Security Considerations

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

7. Acknowledgements

Thhe authors of this document would like to thank Rui Paulo and Michael Scharf for providing valuable comments on earlier versions of this document.

8. References

8.1. Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

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

8.2. Informative References

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

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

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

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

Authors' Addresses

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

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

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

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

TCP Maintenance and Minor Extensions (tcpm)
Internet-Draft
Updates: 793 (if approved)
Intended status: Standards Track
Expires: September 12, 2019

F. Gont
UTN-FRH / SI6 Networks
D. Borman
Quantum Corporation
March 11, 2019

On the Validation of TCP Sequence Numbers
draft-gont-tcpm-tcp-seq-validation-04.txt

Abstract

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

Status of This Memo

Table of Contents

1. Introduction

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

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

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

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. TCP Sequence Number Validation

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

Segment Length	Receive Window	Test
0	0	SEG.SEQ = RCV.NXT
0	>0	RCV.NXT =< SEG.SEQ < RCV.NXT+RCV.WND
>0	0	not acceptable
>0	>0	RCV.NXT =< SEG.SEQ < RCV.NXT+RCV.WND or RCV.NXT =< SEG.SEQ+SEG.LEN-1 < RCV.NXT+RCV.WND

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

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

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

3. Scenarios in which Undesirable Behaviors Might Arise

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

3.1. TCP simultaneous open

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

```
       TCP A                                                     TCP B

    1. CLOSED                                                   CLOSED

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

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

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

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

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

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

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

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

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

(Failed) Simultaneous Connection Synchronization

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

3.2. TCP self connects

Some systems have been found to be unable to process TCP connection requests in which the source endpoint {Source Address, Source Port} is the same as the destination end-point {Destination Address, Destination Port}. Such a scenario might arise e.g. if a process creates a socket, bind()s a local end-point (IP address and TCP port), and then issues a connect() to the same end-point as that specified to bind().

> While not widely employed in existing applications, such a socket could be employed as a "full-duplex pipe" for Inter-Process Communication (IPC).
>
> This scenario is described in detail in pp. 960-962 of [Wright1994].

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

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

3.3. TCP simultaneous close

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

```
       TCP A                                                   TCP B

    1. ESTABLISHED                                          ESTABLISHED

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

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

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

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

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

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

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

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

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

(Failed) Simultaneous Connection Termination

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

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

In line 4, the FIN segment sent by TCP A arrives to TCP B, causing it to transition to the CLOSING state (at this point, TCP B’s RCV.NXT becomes 101). In line 5, TCP A acknowledges the receipt of the TCP B’s FIN segment, and also sets the FIN bit in the outgoing segment (since it has not yet been acknowledged). In line 6, TCP B acknowledges the receipt of TCP A’s FIN segment, and also sets the FIN bit in the outgoing segment (since it has not yet been acknowledged). In line 7, the FIN/ACK from line 5 arrives at TCP B. In line 8, the FIN/ACK from line 6 fails the TCP sequence number validation check, and thus elicits a ACK segment (the segment also contains the FIN bit set, since it had not yet been acknowledged). In line 9, the FIN/ACK from line 7 fails the TCP sequence number validation check, and hence elicits an ACK segment (the segment also contains the FIN bit set, since it had not yet been acknowledged). In line 10, the FIN/ACK from line 8 finally arrives at TCP B.

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

3.4. Simultaneous Window Probes

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

```
   TCP A                                                     TCP B

1. ESTABLISHED                                               ESTABLISHED

2.                       (both TCP windows open)

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

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

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

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

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

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

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

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

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

(Failed) Simultaneous Connection Termination

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

Probe". In line 4, the "persist timer" at TCP B expires, and hence TCP B sends a "Window Probe".

Both Window Probes cross each other in the network.

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

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

4. Updating RFC 793

4.1. TCP sequence number validation

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

---------------- cut here -------------- cut here ----------------

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

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

or

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

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

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

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

   0       0     SEG.SEQ = RCV.NXT

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

  >0       0     not acceptable

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

---------------- cut here -------------- cut here ----------------

is replaced with:

```
 ---------------- cut here -------------- cut here ----------------
```

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

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

or

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

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

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

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

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

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

    >0       0     not acceptable

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

```
 ---------------- cut here -------------- cut here ----------------
```

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

---------------- cut here -------------- cut here ----------------

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

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

Segment Length	Receive Window	Test
0	0	SEG.SEQ = RCV.NXT
0	>0	RCV.NXT =< SEG.SEQ < RCV.NXT+RCV.WND
>0	0	not acceptable
>0	>0	RCV.NXT =< SEG.SEQ < RCV.NXT+RCV.WND or RCV.NXT =< SEG.SEQ+SEG.LEN-1 < RCV.NXT+RCV.WND

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

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

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

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

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

---------------- cut here -------------- cut here ----------------

is replaced with:

---------------- cut here -------------- cut here ----------------

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

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

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

Segment Length	Receive Window	Test
0	0	RCV.NXT-1 =< SEG.SEQ <= RCV.NXT
0	>0	RCV.NXT-1 =< SEG.SEQ < RCV.NXT+RCV.WND
>0	0	not acceptable
>0	>0	RCV.NXT-1 =< SEG.SEQ < RCV.NXT+RCV.WND or RCV.NXT-1 =< SEG.SEQ+SEG.LEN-1 < RCV.NXT+RCV.WND

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

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

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

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

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

```
      one byte to the left of the receive window.
  ---------------- cut here -------------- cut here ----------------
```

4.2. Alternative fix for TCP sequence number validation

The Linux kernel performs a slightly different TCP sequence number validation check, that can accommodate window probes of any size (as opposed to the de facto standard 1-byte window probes). This makes the code more general, at the expense of additional state in the TCB (e.g., the TCP sequence number employed in the last window probe).

4.3. TCP self connects

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

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

5. IANA Considerations

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

6. Security Considerations

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

7. Acknowledgements

Thhe authors of this document would like to thank Theo de Raadt, Rui Paulo and Michael Scharf for providing valuable comments on earlier versions of this document.

8. References

8.1. Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

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

8.2. Informative References

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

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

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

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

Authors' Addresses

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

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

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

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

TSVWG Working Group
Internet-Draft
Intended status: Experimental
Expires: September 4, 2018
L. Han
Y. Qu
Huawei
T. Nadeau
Lucid Vision
March 3, 2018

A New Congestion Control in Bandwidth Guaranteed Network
draft-han-tsvwg-cc-00

Abstract

In bandwidth guaranteed networks, network resources are reserved before a TCP session starts transmitting data. This draft proposes a new TCP congestion control algorithm used in bandwidth guaranteed networks. It is an extension to the current TCP standards.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The original IP protocol suite was designed to support best-effort data transmission. With the development of the Internet, congestion became a real problem. To avoid congestion in the Internet, TCP uses congestion-avoidance algorithms to keep hosts from pumping too much traffic into the network. Over the past 40 years there have been various algorithms and optimizations proposed to solve this problem, including TCP-RENO [RFC5681], TCP-NewReno [RFC6582] [RFC6675], TCP-Cubic [RFC8312] and BBR [I-D.cardwell-iccrg-bbr-congestion-control] etc.

In bandwidth guaranteed networks, network resources are reserved before transmitting data. This draft proposes a new congestion control algorithm that should be used in bandwidth guaranteed networks to improve TCP throughput. The following is a list of key differences between this new algorithm and classic TCP congestion control [RFC5681]:

> It doesn’t have a slow start, after a TCP session is successfully initiated its congestion window (cwnd) jumps to CIR and the host is allowed to transmit data. This is based on the assumption that network resources have been reserved in bandwidth guaranteed networks.

During congestion avoidance, cwnd stays between CIR (Committed Information Rate) and PIR (Peak Information Rate). If there is no packet loss due to congestion, cwnd has a flat top rate as PIR.

OAM is used together with duplicate ACKs to detect whether a packet loss is due to congestion or random failure.

This draft is organized as follows. Section 2 defines terminologies used in this draft. Section 3 provides background information for Bandwidth Guaranteed Networks. Section 4 explains the details of the new congestion control algorithm.

2. Terminology and Notation

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].

Some of the following terms are defined the same as [RFC5681], and they are copied here for readability.

FULL-SIZED SEGMENT: A segment that contains the maximum number of data bytes permitted (i.e., a segment containing SMSS bytes of data).

RECEIVER WINDOW (rwnd): The most recently advertised receiver window.

CONGESTION WINDOW (cwnd): A TCP state variable that limits the amount of data a TCP can send. At any given time, a TCP MUST NOT send data with a sequence number higher than the sum of the highest acknowledged sequence number and the minimum of cwnd and rwnd.

Sender Maximum Segment Size (SMSS): The SMSS is the size of the largest segment that the sender can transmit. This value can be based on the maximum transmission unit of the network, the path MTU discovery [RFC1191, RFC4821] algorithm, RMSS (see next item), or other factors. The size does not include the TCP/IP headers and options.

RECEIVER MAXIMUM SEGMENT SIZE (RMSS): The RMSS is the size of the largest segment the receiver is willing to accept. This is the value specified in the MSS option sent by the receiver during connection startup. Or, if the MSS option is not used, it is 536 bytes [RFC1122]. The size does not include the TCP/IP headers and options.

INITIAL WINDOW (IW): The initial window is the size of the sender's congestion window after the three-way handshake is completed.

RESTART WINDOW (RW): The restart window is the size of the congestion window after a TCP restarts transmission after an idle period.

ssthresh: Slow Start Threshold.

OAM: Operations, Administrations, and Maintenance.

RTT: Round-Trip Time.

CIR: Committed Information Rate.

PIR: Peak Information Rate.

3. Bandwidth Guaranteed Network

With the development of new applications, such as AR/VR, the network is required to provide bandwidth guaranteed services. There have been various solutions, including out-of-band signaling protocols such as RSVP [RFC2205] and NSIS [RFC4080], and in-band-signaling as proposed in [I-D.han-6man-in-band-signaling-for-transport-qos]. The common objective of all these solutions is to have network resources/ bandwidth reserved before data is transmitted. The details of how the resource is reserved are out of the scope of this draft, however it is assumed that in bandwidth guaranteed networks there have been network resources (bandwidths, queues etc.) dedicated to the TCP flows, and data is guaranteed at CIR rate. When data rate is between CIR and PIR shared resources are used, and traffic above CIR rate is not guaranteed. No traffic above PIR rate will be allowed to enter the network.

The proposed congestion control also requires that OAM (Operations, administration and management) is used to constantly report on the network condition parameters. Before a TCP session is started, important network parameters need to be detected by OAM, such as number of hops, Round Trip Time (RTT). This might be done through setting up a measuring TCP connection. The measuring TCP connection does not have user data, and it is only used to measure the key network parameters. As the network status is constantly changing, after a TCP session is established, these parameters need to be updated. This requires a sender to periodically or consistently embed TCP data packet with OAM [I-D.han-6man-in-band-signaling-for-transport-qos] [I-D.ietf-ippm-ioam-data] to detect current buffer depth, RTT etc.

It is important that OAM needs to be able to detect if any device's buffer depth has exceeded the pre-configured threshold, as this is an indication of potential congestion and packet drop. When this happens, OAM should send a possible congestion alarm to the TCP sender. In case the retransmit timer expires on this TCP sender, if a possible congestion alarm has been received it means a packet is dropped due to congestion. Otherwise it is possible that this packet drop might due to some physical failure. The OAM details are out of the scope of this draft. Please refer to other related drafts.

In summary, in bandwidth guaranteed networks resources are reserved before transmitting data, and OAM is used to get network statistics. The new congestion control proposed in this draft is to be used in this kind of bandwidth guaranteed networks.

4. New Congestion Control

[RFC5681] defines a set of TCP congestion algorithms: slow start, congestion avoidance, fast retransmit and fast recovery. The proposed congestion control in this draft is an extension to RFC 5681, and it only differs in the congestion control algorithm on the sender side.

4.1. Receiver Advertised Window Size

Receiver's advertised window (rwnd) is a receiver-side limit on the amount of outstanding data, so a sender should not send data more than this window size. It is calculated as the following:

```
rwnd = AdvertisedWND = MaxRcvBuffer - (LastByteRcvd - LastByteRead)
```

4.2. MinBandwidthWND and MaxBandwidthWND

Same as [RFC5681], on the sender side, the congestion window (cwnd) is the sender-side limit on the amount of data that the sender can transmit before receiving an acknowledgement (ACK). Considering both the sender and the receiver side, the effective sending window is always the minimum of cwnd and rwnd:

```
EffectiveWND = min(cwnd, rwnd)
```

A TCP sender MUST NOT send data more than the minimum of cwnd and rwnd.

Slow-start is commonly used in TCP at the beginning of a transfer or after a loss repair as the network conditions are unknown, hence this slow probing is necessary to determine the available network capacity in order to avoid inappropriately sending large burst of data into the network and cause congestion. A detailed discussion about initial window setting is provided in [RFC3390].

RTT is the time taken to send a packet to the destination plus receiving a response packet(ACK). Since the network status is constantly changing, RTT also varies. [RFC6298] specifies how RTT should be sampled and updated. In this new algorithm RTT is updated using the following formula:

```
RTT = a* old RTT + (1-a) * new RTT   (0 < a < 1)   (1)
```

The initial RTT can be achieved using a measure TCP connection, or configured based on historical data.

In bandwidth guaranteed network since resources are already allocated and the network status is known through OAM [I-D.han-6man-in-band-signaling-for-transport-qos], it is safe to remove slow-start and allow a host to start sending traffic at the rate of CIR after the TCP session is established.

There are two important window sizes, the MinBandwidthWND and the MaxBandwidthWND are calculated as below:

```
MinBandwidthWND = CIR * RTT/MSS    (2)
MaxBandwidthWND = PIR * RTT/MSS    (3)
```

In bandwidth guaranteed networks, after a TCP session is established, the sender can start transmitting data at an initial window size, which is equal to MinBandwidthWND:

```
cwnd = MinBandwidthWND
IW = min (cwnd, rwnd)
```

If the receiver window (rwnd) is not a limiting factor, the sender will start sending data at CIR rate. This is a key difference from the classic TCP slow-start, which usually starts from sending one or two packets [RFC5681].

4.3. Congestion Avoidance

In TCP-Reno, a TCP enters congestion avoidance mode after slow-start. In bandwidth guaranteed networks, there is no slow-start, so a TCP enters congestion avoidance mode right after the initial start.

During congestion avoidance, for approximately per round-trip time when a valid ACK packet is received, cwnd is increased by one until it reaches MaxBandwidthWND.

```
If (cwnd < MaxBandwidthWND) {
  cwnd +=1;
} else {
  cwnd = MaxBandwidthWND;
}
```

Once the cwnd reaches MaxBandwidthWND , it stays constant at MaxBandwidthWND until packet loss is detected. This is another major difference from [RFC5681]. In [RFC5681] congestion avoidance period, the cwnd keeps increasing until a TCP sender detects segment loss. However, in this new congestion control algorithm, the cwnd stays constant at MaxBandwidthWND until there is packet loss detected.

This means a TCP sender is never allowed to send data at a rate larger than PIR, and it’s different from TCP Reno.

4.4. Fast Retransmit and Fast Recovery

Same as defined [RFC5681], a TCP receiver SHOULD send an immediate duplicate ACK when an out-of-order segment arrives. The TCP sender detects and repair loss based on incoming duplicate ACKs. If 3 duplicate ACKs are received, the sender uses it as an indication that a segment has been lost, and will perform a retransmission of the lost segment.

In TCP-Reno [RFC5681], after the fast retransmit of what appears to be the lost segment, fast recovery is used to continue to transmit new segments at a reduced rate ssthresh.

In the new congestion control algorithm, upon receiving duplicate ACKs the fast retransmit and fast recovery follow the below rules:

- When a sender receives the first and second duplicate ACKs, same as [RFC5681], the cwnd is not changed, and the sender continues to send traffic.

- When a sender receives the third duplicated ACK, if the retransmission timer has not expired and a previous OAM congestion alarm has been received it is likely a segment is lost due to congestion. The sender will perform a retransmission of the lost segment, and the cwnd is set to be MinBandwidthWND.

- When a sender receives the third duplicated ACK, but no previous OAM congestion alarm has been received, then it is considered that a segment is lost due to random failure not congestion. In this case the cwnd is not changed.

Compared to [RFC5681], where in case of network congestion the new cwnd is set to be ssthresh, which is usually half of the old cwnd. In this new congestion control, in case there is a segment loss detected as described above, the new cwnd is set to be MinBandwithWND as in equation (2).

4.5. Timeout

If a retransmission timer [RFC6298] in a TCP sender expires, in bandwidth guaranteed networks no matter duplicate ACK received or not, this most likely indicates a physical failure.

In this case, the cwnd is set to be one, and the TCP sender will retransmit the lost segment. This packet also services the function of probing network status. If there is really a network failure, no ACK will be received and the retransmission timer will expire again. Upon receiving an expected ACK after the retransmission, it means the network has recovered, and the cwnd will be set to be MinBandwidthWND as in equation (2).

4.6. Idle Recovery

It is defined in [RFC5681] that a TCP session should use slow start to restart transmission after a long idle period more than one retransmission timeout, and the RW (Restart Window) is the minimum of IW and cwnd.

In this proposal, the same rule is still followed. However due to the fact that there is no slow start needed in bandwidth guaranteed networks, and the IW in this new congestion control is set to be MinBandwidthWND, a TCP sender can start transmitting data at CIR rate after a long idle.

5. IANA Considerations

NA.

6. Security Considerations

This proposal makes no change to the underlying security of TCP. More information about TCP security concerns can be found in [RFC5681].

7. References

7.1. 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, <https://www.rfc-editor.org/info/rfc2119>.

7.2. Informative References

[RFC2205] Braden, R., Ed., Zhang, L., Berson, S., Herzog, S., and S. Jamin, "Resource ReSerVation Protocol (RSVP) -- Version 1 Functional Specification", RFC 2205, DOI 10.17487/RFC2205, September 1997, <https://www.rfc-editor.org/info/rfc2205>.

[RFC3390] Allman, M., Floyd, S., and C. Partridge, "Increasing TCP's Initial Window", RFC 3390, DOI 10.17487/RFC3390, October 2002, <https://www.rfc-editor.org/info/rfc3390>.

[RFC4080] Hancock, R., Karagiannis, G., Loughney, J., and S. Van den Bosch, "Next Steps in Signaling (NSIS): Framework", RFC 4080, DOI 10.17487/RFC4080, June 2005, <https://www.rfc-editor.org/info/rfc4080>.

[RFC4960] Stewart, R., Ed., "Stream Control Transmission Protocol", RFC 4960, DOI 10.17487/RFC4960, September 2007, <https://www.rfc-editor.org/info/rfc4960>.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <https://www.rfc-editor.org/info/rfc5681>.

[RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, DOI 10.17487/RFC6298, June 2011, <https://www.rfc-editor.org/info/rfc6298>.

[RFC6582] Henderson, T., Floyd, S., Gurtov, A., and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 6582, DOI 10.17487/RFC6582, April 2012, <https://www.rfc-editor.org/info/rfc6582>.

[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, DOI 10.17487/RFC6675, August 2012, <https://www.rfc-editor.org/info/rfc6675>.

[RFC8312] Rhee, I., Xu, L., Ha, S., Zimmermann, A., Eggert, L., and R. Scheffenegger, "CUBIC for Fast Long-Distance Networks", RFC 8312, DOI 10.17487/RFC8312, February 2018, <https://www.rfc-editor.org/info/rfc8312>.

[I-D.cardwell-iccrg-bbr-congestion-control]
Cardwell, N., Cheng, Y., Yeganeh, S., and V. Jacobson, "BBR Congestion Control", draft-cardwell-iccrg-bbr-congestion-control-00 (work in progress), July 2017.

[I-D.han-6man-in-band-signaling-for-transport-qos]
Han, L., Li, G., Tu, B., Xuefei, T., Li, F., Li, R., Tantsura, J., and K. Smith, "IPv6 in-band signaling for the support of transport with QoS", draft-han-6man-in-band-signaling-for-transport-qos-00 (work in progress), October 2017.

[I-D.ietf-ippm-ioam-data]
Brockners, F., Bhandari, S., Pignataro, C., Gredler, H., Leddy, J., Youell, S., Mizrahi, T., Mozes, D., Lapukhov, P., Chang, R., and d. daniel.bernier@bell.ca, "Data Fields for In-situ OAM", draft-ietf-ippm-ioam-data-01 (work in progress), October 2017.

Acknowledgments

The authors wish to thank xxxx for their helpful comments and suggestions.

Authors' Addresses

Lin Han
Huawei
2330 Central Expressway
Santa Clara CA 95050
USA

EMail: lin.han@huawei.com

Yingzhen Qu
Huawei
2330 Central Expressway
Santa Clara CA 95050
USA

EMail: yingzhen.qu@huawei.com

Thomas Nadeau
Lucid Vision
Hampton NH 03842
USA

EMail: tnadeau@lucidvision.com

TCP Maintenance & Minor Extensions (tcpm)
Internet-Draft
Intended status: Experimental
Expires: September 6, 2018
B. Briscoe
CableLabs
M. Kuehlewind
ETH Zurich
R. Scheffenegger
March 5, 2018

More Accurate ECN Feedback in TCP
draft-ietf-tcpm-accurate-ecn-06

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,ew 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

Copyright Notice

Table of Contents

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 [RFC7713]), DCTCP [RFC8257] or L4S [I-D.ietf-tsvwg-l4s-arch] need more accurate ECN feedback information than provided by the feedback scheme as specified in [RFC3168] whenever more than one marking is received in one RTT. This document specifies an alternative feedback scheme that provides more accurate information and could be used by these new TCP extensions. 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 AccECN for short. If AccECN progresses from experimental to the standards track, it is intended to be a complete replacement for classic TCP/ECN feedback, not a fork in the design of TCP. AccECN feedback complements TCP's loss feedback and it supplements classic TCP/ECN feedback, so its applicability is intended to include all public and private IP networks (and even any non-IP networks over which TCP is used today), whether or not any nodes on the path support ECN of whatever flavour.

Until the AccECN experiment succeeds, [RFC3168] will remain as the only 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].

AccECN feedback overloads the two existing ECN flags as well as the currently reserved and previously called NS flag 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.

AccECN is solely an (experimental) change to the TCP wire protocol; it only specifies the negotiation and signaling of more accurate ECN feedback from a TCP Data Receiver to a Data Sender. It is completely independent of how TCP might respond to congestion feedback, which is out of scope. For that we refer to [RFC3168] or any RFC that specifies a different response to TCP ECN feedback, for example: [RFC8257]; or the ECN experiments referred to in [RFC8311], namely: a TCP-based Low Latency Low Loss Scalable (L4S) congestion control [I-D.ietf-tsvwg-l4s-arch]; ECN-capable TCP control packets [I-D.ietf-tcpm-generalized-ecn], or Alternative Backoff with ECN (ABE) [I-D.ietf-tcpm-alternativebackoff-ecn].

It is likely (but not required) that the AccECN protocol will be implemented along with the following experimental additions to the TCP-ECN protocol: ECN-capable TCP control packets and retransmissions [I-D.ietf-tcpm-generalized-ecn], which includes the ECN-capable SYN/ACK experiment [RFC5562]; and testing receiver non-compliance [I-D.moncaster-tcpm-rcv-cheat].

1.1. Document Roadmap

The following introductory sections outline the goals of AccECN (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 AccECN protocol. Then Section 3 gives the normative protocol specification. Section 4 assesses the interaction of AccECN with commonly used variants of TCP, whether standardised or not. Section 5 summarises the features and properties of AccECN.

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 AccECN 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 AccECN 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 is deployed and 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 (e.g. hardware offloading and middleboxes), whether or not they comply with standards.

Testing will mostly focus on fall-back strategies in case of middlebox interference. Current recommended strategies are specified in Sections 3.1.2, 3.2.3, 3.2.4 and 3.2.7. The effectiveness of these strategies depends on the actual deployment situation of middleboxes. Therefore experimental verification to confirm large-scale path traversal in the Internet is needed before finalizing this specification on the Standards Track.

Another experimentation focus is the implementation feasibiliy of change-triggered ACKs as described in section 3.2.8. While on average this should not lead to a higher ACK rate, it changes the ACK patter which especially can have an impact on hardware offload. Further experimentation is needed to advise if this should a hard requirement or just prefer behavior.

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 BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

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 last bit in byte 13 of the TCP header was defined as the Nonce Sum (NS) for the ECN Nonce [RFC3540]. In the absence of widespread deployment RFC 3540 has been reclassified as historic [RFC8311] and the respective flag has been marked as "reserved", making this TCP flag available for use by the AccECN experiment instead.

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

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

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 (see Figure 2 later). 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. There is no need to acknowledge these continually repeated counters, so the congestion window reduced (CWR) mechanism is no longer used. 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 cyling 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 [RFC8311]. 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 AccECN 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 AccECN 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, AccECN 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 AccECN provides feedback on whether it is CE marked. Although RFC 3168 prohibits an ECN-capable SYN, providing feedback of CE marking on the SYN supports future scenarios in which SYNs might be ECN-enabled (without prejudging whether they ought to be). For instance, [RFC8311] updates this aspect of RFC 3168 to allow experimentation with ECN-capable TCP control packets.

Even if the TCP client (or server) has set the SYN (or SYN/ACK) to not-ECT in compliance with RFC 3168, feedback on the state of the ECN field when it arrives at the receiver could still be useful, because middleboxes have been known to overwrite the ECN IP field as if it is still part of the old Type of Service (ToS) field [Mandalari18]. 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]). Today, if a TCP server receives ECT or 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, prior to AccECN, the server's only safe course of action was to disable ECN for the connection. Instead, the AccECN protocol allows the server to feed back the received ECN field to the client, which then has all the information to decide whether the connection has to fall-back from supporting ECN (or not).

3. AccECN Protocol Specification

3.1. Negotiating to use AccECN

3.1.1. Negotiation during the TCP handshake

Given the ECN Nonce [RFC3540] has been reclassified as historic [RFC8311], the present specification renames the TCP flag at bit 7 of the TCP header flags from NS (Nonce Sum) to AE (Accurate ECN) (see IANA Considerations in Section 6).

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 AE=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 TCP flags on the SYN/ACK to one of the 4 values shown in the top block of Table 2 to confirm that it supports AccECN. The TCP server MUST NOT set one of these 4 combination of flags on the SYN/ACK unless the preceding SYN requested support for AccECN as above.

A TCP server in AccECN mode MUST set the AE, CWR and ECE TCP flags on the SYN/ACK to the value in Table 2 that feeds back the IP-ECN field that arrived on the SYN. This applies whether or not the server itself supports setting the IP-ECN field on a SYN or SYN/ACK (see Section 2.5 for rationale).

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.

The procedure for the client to follow if a SYN/ACK does not arrive before its retransmission timer expires is given in Section 3.1.2.

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. The entries in the first two columns have been abbreviated, as follows:

AccECN: More Accurate ECN Feedback (the present specification)

Nonce: ECN Nonce feedback [RFC3540]

ECN: 'Classic' ECN feedback [RFC3168]

No ECN: Not-ECN-capable. Implicit congestion notification using packet drop.

A	B	SYN A->B	SYN/ACK B->A	Feedback Mode
		AE CWR ECE	AE CWR ECE	
AccECN	AccECN	1 1 1	0 1 0	AccECN (Not-ECT on SYN)
AccECN	AccECN	1 1 1	0 1 1	AccECN (ECT1 on SYN)
AccECN	AccECN	1 1 1	1 0 0	AccECN (ECT0 on SYN)
AccECN	AccECN	1 1 1	1 1 0	AccECN (CE on SYN)
AccECN	Nonce	1 1 1	1 0 1	classic ECN
AccECN	ECN	1 1 1	0 0 1	classic ECN
AccECN	No ECN	1 1 1	0 0 0	Not ECN
Nonce	AccECN	0 1 1	0 0 1	classic ECN
ECN	AccECN	0 1 1	0 0 1	classic ECN
No ECN	AccECN	0 0 0	0 0 0	Not ECN
AccECN	Broken	1 1 1	1 1 1	Not ECN

Table 2: ECN capability negotiation between Client (A) and Server (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 a combination labelled ‘Broken’ . Some older TCP server implementations incorrectly set the reserved flags in the SYN/ACK by reflecting those in the SYN. Such broken TCP servers (B) cannot support ECN, so as soon as an AccECN-capable TCP client (A) receives such a broken SYN/ACK it MUST fall-back to Not ECN mode for both its half connections.

The following exceptional cases need some explanation:

ECN Nonce: With AccECN implementation, there is no need for the ECN Nonce feedback mode [RFC3540], which has also been reclassified as historic [RFC8311], as 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 AE=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.

3.1.2. Retransmission of the SYN

If the sender of an AccECN SYN times out before receiving the SYN/ACK, the sender SHOULD attempt to negotiate the use of AccECN at least one more time by continuing to set all three TCP ECN flags on the first retransmitted SYN (using the usual retransmission time-outs). If this first retransmission also fails to be acknowledged, the sender SHOULD send subsequent retransmissions of the SYN without any TCP-ECN flags set. This adds delay, in the case where a middlebox drops an AccECN (or ECN) SYN deliberately. However, current measurements imply that a drop is less likely to be due to middlebox interference than other intermittent causes of loss, e.g. congestion, wireless interference, etc.

Implementers MAY use other fall-back strategies if they are found to be more effective (e.g. attempting to negotiate AccECN on the SYN only once or more than twice (most appropriate during high levels of congestion); or falling back to classic ECN feedback rather than non-

ECN). Further it may 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. In any case, the TCP initiator SHOULD cache failed connection attempts. If it does, it SHOULD NOT give up attempting to negotiate AccECN on the SYN of subsequent connection attempts until it is clear that the blockage is persistently and specifically due to AccECN. The cache should be arranged to expire so that the initiator will infrequently attempt to check whether the problem has been resolved.

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 3.2.7.

3.2. AccECN Feedback

Each Data Receiver of each half connection maintains four counters, r.cep, r.ceb, r.e0b and r.e1b. 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.e1b 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 initializes its counters to r.cep = 5, r.e0b = 1 and r.ceb = r.e1b.= 0 (see Appendix A.5). Non-zero initial values are used to support a stateless handshake (see Section 4.1) and to be distinct from cases where the fields are incorrectly zeroed (e.g. by middleboxes - see Section 3.2.7.4).

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 3.2.6. 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. Therefore the feedback carried on a retransmitted packet is unlikely to be the same as the feedback on the original packet.

3.2.1. Initialization of Feedback Counters at the Data Sender

Each Data Sender of each half connection maintains four counters, s.cep, s.ceb, s.e0b and s.e1b intended to track the equivalent counters at the Data Receiver. When a host enters AccECN mode, it initializes them to s.cep = 5, s.e0b = 1 and s.ceb = s.e1b.= 0.

If a TCP client (A) in AccECN mode receives a SYN/ACK with CE feedback, i.e. AE=1, CWR=1, ECE=0, it increments s.cep to 6. Otherwise, for any of the 3 other combinations of the 3 ECN TCP flags (the top 3 rows in Table 2), s.cep remains initialized to 5.

3.2.2. The ACE Field

After AccECN has been negotiated on the SYN and SYN/ACK, both hosts overload the three TCP flags (AE, CWR and ECE) 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
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|               |           |           | U | A | P | R | S | F |
| Header Length | Reserved  |    ACE    | 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 to ACE unconditionally; it merely overloads them with another name and definition once an AccECN connection has been established.

A host MUST interpret the AE, CWR and ECE flags as the 3-bit ACE counter on a segment with the SYN flag cleared (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).

With only one exception, on any packet with the SYN flag cleared (SYN=0), the Data Receiver MUST encode the three least significant bits of its r.cep counter into the ACE field it feeds back to the Data Sender.

There is only one exception to this rule: On the final ACK of the 3WHS, a TCP client (A) in AccECN mode MUST use the ACE field to feed back which of the 4 possible values of the IP-ECN field were on the SYN/ACK (the binary encoding is the same as that used on the SYN/ACK). Table 3 shows the meaning of each possible value of the ACE field on the ACK of the SYN/ACK and the value that an AccECN server MUST set s.cep to as a result. The encoding in Table 3 is solely applicable on a packet in the client-server direction with an acknowledgement number 1 greater than the Initial Sequence Number (ISN) that was used by the server.

ACE on ACK of SYN/ACK	IP-ECN codepoint on SYN/ACK inferred by server	Initial s.cep of server in AccECN mode
0b000	{Notes 1, 2}	Disable ECN
0b001	{Notes 2, 3}	5
0b010	Not-ECT	5
0b011	ECT(1)	5
0b100	ECT(0)	5
0b101	Currently Unused {Note 3}	5
0b110	CE	6
0b111	Currently Unused {Note 3}	5

Table 3: Meaning of the ACE field on the ACK of the SYN/ACK

{Note 1}: If the server is in AccECN mode, the value of zero raises suspicion of zeroing of the ACE field on the path (see Section 3.2.3).

{Note 2}: If a server is in AccECN mode, there ought to be no valid case where the ACE field on the last ACK of the 3WHS has a value of 0b000 or 0b001.

However, in the case where a server that implements AccECN is also using a stateless handshake (termed a SYN cookie) it will not remember whether it entered AccECN mode. Then these two values remind it that it did not enter AccECN mode (see Section 4.1 for details).

{Note 3}: If the server is in AccECN mode, these values are Currently Unused but the AccECN server’s behaviour is still defined for forward compatibility.

3.2.3. Testing for Zeroing of the ACE Field

Section 3.2.2 required the Data Receiver to initialize the r.cep counter to a non-zero value. Therefore, in either direction the initial value of the ACE field ought to be non-zero.

If AccECN has been successfully negotiated, the Data Sender SHOULD check the initial value of the ACE field in the first arriving segment with SYN=0. If the initial value of the ACE field is zero (0b000), the Data Sender MUST disable sending ECN-capable packets for the remainder of the half-connection by setting the IP/ECN field in all subsequent packets to Not-ECT.

For example, the server checks the ACK of the SYN/ACK or the first data segment from the client, while the client checks the first data segment from the server. More precisely, the "first segment with SYN=0" is defined as: the segment with SYN=0 that i) acknowledges sequence space at least covering the initial sequence number (ISN) plus 1; and ii) arrives before any other segments with SYN=0 so it is unlikely to be a retransmission. If no such segment arrives (e.g. because it is lost and the ISN is first acknowledged by a subsequent segment), no test for invalid initialization can be conducted, and the half-connection will continue in AccECN mode.

Note that the Data Sender MUST NOT test whether the arriving counter in the initial ACE field has been initialized to a specific valid value - the above check solely tests whether the ACE fields have been incorrectly zeroed. This allows hosts to use different initial values as an additional signalling channel in future.

3.2.4. Testing for Mangling of the IP/ECN Field

The value of the ACE field on the SYN/ACK indicates the value of the IP/ECN field when the SYN arrived at the server. The client can compare this with how it originally set the IP/ECN field on the SYN. If this comparison implies an unsafe transition of the IP/ECN field, for the remainder of the connection the client MUST NOT send ECN-capable packets, but it MUST continue to feed back any ECN markings on arriving packets.

The value of the ACE field on the last ACK of the 3WHS indicates the value of the IP/ECN field when the SYN/ACK arrived at the client. The server can compare this with how it originally set the IP/ECN field on the SYN/ACK. If this comparison implies an unsafe transition of the IP/ECN field, for the remainder of the connection the server MUST NOT send ECN-capable packets, but it MUST continue to feedback any ECN markings on arriving packets.

The ACK of the SYN/ACK is not reliably delivered (nonetheless, the count of CE marks is still eventually delivered reliably). If this ACK does not arrive, the server has to continue to send ECN-capable packets without having tested for mangling of the IP/ECN field on the SYN/ACK. Experiments with AccECN deployment will assess whether this limitation has any effect in practice.

Invalid transitions of the IP/ECN field are defined in [RFC3168] and repeated here for convenience:

- the not-ECT codepoint changes;
- either ECT codepoint transitions to not-ECT;
- the CE codepoint changes.

RFC 3168 says that a router that changes ECT to not-ECT is invalid but safe. However, from a host's viewpoint, this transition is unsafe because it could be the result of two transitions at different routers on the path: ECT to CE (safe) then CE to not-ECT (unsafe). This scenario could well happen where an ECN-enabled home router congests its upstream mobile broadband bottleneck link, then the ingress to the mobile network clears the ECN field [Mandalari18].

The above fall-back behaviours are necessary in case mangling of the IP/ECN field is asymmetric, which is currently common over some mobile networks [Mandalari18]. Then one end might see no unsafe transition and continue sending ECN-capable packets, while the other end sees an unsafe transition and stops sending ECN-capable packets.

3.2.5. 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.

3.2.6. 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.e1b counters, respectively. The initial 'E' of each field name stands for 'Echo'.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Kind = TBD1  |  Length = 11  |          EE0B field           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| EE0B (cont'd) |           ECEB field                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  EE1B field                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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 3.2.8 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. If the AccECN Option is of any other length, implementations MUST use those whole 3 octet fields that fit within the length and ignore the remainder of the option.

The use of the AccECN option is optional for the Data Receiver. If the Data Receiver intents to use the AccECN option at any time during the rest of the connection it strongly recommended to also test its path traversal by including it in the SYN/ACK as specified in the next section. By default the use of the AccECN option is RECOMMENDED.

3.2.7. Path Traversal of the AccECN Option

3.2.7.1. Testing the AccECN Option during the Handshake

The TCP client 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. A fall-back strategy for the loss of the SYN (possibly due to middlebox interference) is specified in Section 3.1.2.

A TCP server that confirms its support for AccECN (in response to an AccECN SYN from the client as described in Section 3.1) SHOULD 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 3WHS. 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 MAY 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.

3.2.7.2. Testing for Loss of Packets Carrying the AccECN Option

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 TCP server SHOULD retransmit the SYN/ACK with the same TCP flags (AE, CWR and ECE) but with no AccECN Option. If this retransmission times out, to expedite connection setup, the TCP server SHOULD disable AccECN and ECN for this connection by retransmitting the SYN/ACK with AE=CWR=ECE=0 and no AccECN Option. Implementers MAY use other fall-back strategies if they are found to be more effective (e.g. falling back to classic ECN feedback on the first retransmission; retrying the AccECN Option for a second time before fall-back (most appropriate during high levels of congestion); or falling back to classic ECN feedback rather than non-ECN on the third retransmission).

If the TCP client detects that the first data segment it sent with the AccECN Option 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 whether the AccECN Option is blocked for subsequent connections.

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

If the TCP server receives a second SYN with a request for AccECN support, it should resend the SYN/ACK, again confirming its support for AccECN, but this time without the AccECN Option. This approach rules out any interference by middleboxes that may drop packets with unknown options, even though it is more likely that the SYN/ACK would have been lost due to congestion. The TCP server MAY try to send another packet with the AccECN Option at a later point during the connection but should monitor if that packet got lost as well, in which case it SHOULD disable the sending of the AccECN Option for this half-connection.

Similarly, an AccECN end-point MAY separately memorize which data packets carried an AccECN Option and disable the sending of AccECN Options if the loss probability of those packets is significantly higher than that of all other data packets in the same connection.

3.2.7.3. Testing for Stripping of the 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 segment that acknowledges sequence space at least covering the ISN, it switches into a mode that assumes that the AccECN Option is not available for this half connection.

While a host is in this mode that assumes incoming AccECN Options are not available, it MUST adopt the conservative interpretation of the ACE field discussed in Section 3.2.5. However, it cannot make any assumption about support of outgoing AccECN Options on the other half connection, so it SHOULD continue to send the AccECN Option itself (unless it has established that sending the AccECN Option is causing packets to be blocked as in Section 3.2.7.2).

If a host is in the mode that assumes incoming AccECN Options are not available, but it receives an AccECN Option at any later point during the connection, this clearly indicates that the AccECN Option is not blocked on the respective path, and the AccECN endpoint MAY switch out of the mode that assumes the AccECN Option is not available for this half connection.

3.2.7.4. Test for Zeroing of the AccECN Option

For a related test for invalid initialization of the ACE field, see Section 3.2.3

Section 3.2 required the Data Receiver to initialize the r.e0b counter to a non-zero value. Therefore, in either direction the initial value of the EE0B field in the AccECN Option (if one exists) ought to be non-zero. If AccECN has been negotiated:

o the TCP server MAY check the initial value of the EE0B field in the first segment that acknowledges sequence space that at least covers the ISN plus 1. If the initial value of the EE0B field is zero, the server will switch into a mode that ignores the AccECN Option for this half connection.

o the TCP client MAY check the initial value of the EE0B field on the SYN/ACK. If the initial value of the EE0B field is zero, the client will switch into a mode that ignores the AccECN Option for this half connection.

While a host is in the mode that ignores the AccECN Option it MUST adopt the conservative interpretation of the ACE field discussed in Section 3.2.5.

Note that the Data Sender MUST NOT test whether the arriving byte counters in the initial AccECN Option have been initialized to specific valid values - the above checks solely test whether these fields have been incorrectly zeroed. This allows hosts to use different initial values as an additional signalling channel in future. Also note that the initial value of either field might be greater than its expected initial value, because the counters might already have been incremented. Nonetheless, the initial values of the counters have been chosen so that they cannot wrap to zero on these initial segments.

3.2.7.5. Consistency between AccECN Feedback Fields

When the AccECN Option is available it supplements but does not replace the ACE field. An endpoint using AccECN feedback MUST always consider the information provided in the ACE field whether or not the AccECN Option is also available.

If the AccECN option is present, the s.cep counter might increase while the s.ceb counter does not (e.g. due to a CE-marked control packet). The sender’s response to such a situation is out of scope, and needs to be dealt with in a specification that uses ECN-capable control packets. Theoretically, this situation could also occur if a middlebox mangled the AccECN Option but not the ACE field. However, the Data Sender has to assume that the integrity of the AccECN Option is sound, based on the above test of the well-known initial values and optionally other integrity tests (Section 4.3).

If either end-point detects that the s.ceb counter has increased but the s.cep has not (and by testing ACK coverage it is certain how much the ACE field has wrapped), this invalid protocol transition has to be due to some form of feedback mangling. So, the Data Sender MUST disable sending ECN-capable packets for the remainder of the half-connection by setting the IP/ECN field in all subsequent packets to Not-ECT.

3.2.8. 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 MUST immediately send an ACK with an AccECN Option, without waiting for the next delayed ACK (this is in addition to the safety recommendation in Section 3.2.5 against ambiguity of the ACE field).

This is stated as a "MUST" so that the data sender can rely on change-triggered ACKs to detect transitions right from the very start of a flow, without first having to detect whether the receiver complies. A concern has been raised that certain offload hardware needed for high performance might not be able to support change-triggered ACKs, although high performance protocols such as DCTCP successfully use change-triggered ACKs. One possible experimental compromise would be for the receiver to heuristically detect whether the sender is in slow-start, then to implement change-triggered ACKs in software while the sender is in slow-start, and offload to hardware otherwise. If the operator disables change-triggered ACKs, whether partially like this or otherwise, the operator will also be responsible for ensuring a co-ordinated sender algorithm is deployed;

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 3.2.6);

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 IP/ECN fields 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 is deliberately worded to ignore the arrival of a control packet with no payload, which therefore does not alter any byte counters, because it is important that TCP does not acknowledge pure ACKs. 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. Requirements for TCP Proxies, Offload Engines and other Middleboxes on AccECN Compliance

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 intervenes 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 SHOULD preserve the timing of each ACK (for example, if it coalesced ACKs it would not be AccECN-compliant) as these can be used by the Data Sender to infer further information about the path congestion level. 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 3.2.6, 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 SHOULD also 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 it receives a pure ACK that acknowledges an ISN that is a valid SYN cookie, and if the ACK contains an ACE field with the value 0b010 to 0b111 (decimal 2 to 7), 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.

If the pure ACK that acknowledges a SYN cookie contains an ACE field with the value 0b000 or 0b001, these values indicate that the client did not request support for AccECN and therefore the server does not enter AccECN mode for this connection. Further, 0b001 on the ACK implies that the server sent an ECN-capable SYN/ACK, which was marked CE in the network, and the non-AccECN client fed this back by setting ECE on the ACK of the SYN/ACK.

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 3.2.8 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

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 [RFC3540], 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 networks 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 [RFC7713]. 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.

Originally the ECN Nonce [RFC3540] was proposed to ensure integrity of congestion feedback. With minor changes AccECN could be optimised for the possibility that the ECT(1) codepoint might be used as an ECN Nonce. However, given RFC 3540 has been reclassified as historic, 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.

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 3WHS 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 reassigns bit 7 of the TCP header flags to the AccECN experiment. This bit was previously called the Nonce Sum (NS) flag [RFC3540], but RFC 3540 is being reclassified as historic [RFC8311]. The flag will now be defined as:

Bit	Name	Reference
7	AE (Accurate ECN)	RFC XXXX

[TO BE REMOVED: This registration should take place at the following location: https://www.iana.org/assignments/tcp-header-flags/tcp-header-flags.xhtml#tcp-header-flags-1]

This document also 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), 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 3.2.5). 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 3.2.5 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. AccECN is compatible with the three 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.

There is a potential 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.

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, Michael Welzl, Gorry Fairhurst, David Black, Spencer Dawkins, Michael Scharf and Michael Tuexen 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). He was also part-funded by the Research Council of Norway through the TimeIn project. The views expressed here are solely those of the authors.

Mirja Kuehlewind was partly supported by the European Commission under Horizon 2020 grant agreement no. 688421 Measurement and Architecture for a Middleboxed Internet (MAMI), and by the Swiss State Secretariat for Education, Research, and Innovation under contract no. 15.0268. This support does not imply endorsement.

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, <https://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, <https://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, <https://www.rfc-editor.org/info/rfc5681>.

[RFC6994] Touch, J., "Shared Use of Experimental TCP Options", RFC 6994, DOI 10.17487/RFC6994, August 2013, <https://www.rfc-editor.org/info/rfc6994>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

10.2. Informative References

[I-D.ietf-tcpm-alternativebackoff-ecn]
Khademi, N., Welzl, M., Armitage, G., and G. Fairhurst, "TCP Alternative Backoff with ECN (ABE)", draft-ietf-tcpm-alternativebackoff-ecn-06 (work in progress), February 2018.

[I-D.ietf-tcpm-generalized-ecn]
Bagnulo, M. and B. Briscoe, "ECN++: Adding Explicit Congestion Notification (ECN) to TCP Control Packets", draft-ietf-tcpm-generalized-ecn-02 (work in progress), October 2017.

[I-D.ietf-tsvwg-l4s-arch]
Briscoe, B., Schepper, K., and M. Bagnulo, "Low Latency, Low Loss, Scalable Throughput (L4S) Internet Service: Architecture", draft-ietf-tsvwg-l4s-arch-01 (work in progress), October 2017.

[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.

[Mandalari18]
Mandalari, A., Lutu, A., Briscoe, B., Bagnulo, M., and Oe. Alay, "Measuring ECN++: Good News for ++, Bad News for ECN over Mobile", IEEE Communications Magazine , March 2018.

(to appear)

[RFC3540] Spring, N., Wetherall, D., and D. Ely, "Robust Explicit Congestion Notification (ECN) Signaling with Nonces", RFC 3540, DOI 10.17487/RFC3540, June 2003, <https://www.rfc-editor.org/info/rfc3540>.

[RFC4987] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", RFC 4987, DOI 10.17487/RFC4987, August 2007, <https://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, <https://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, <https://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, <https://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, <https://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, <https://www.rfc-editor.org/info/rfc7560>.

[RFC7713] Mathis, M. and B. Briscoe, "Congestion Exposure (ConEx) Concepts, Abstract Mechanism, and Requirements", RFC 7713, DOI 10.17487/RFC7713, December 2015, <https://www.rfc-editor.org/info/rfc7713>.

[RFC8257] Bensley, S., Thaler, D., Balasubramanian, P., Eggert, L., and G. Judd, "Data Center TCP (DCTCP): TCP Congestion Control for Data Centers", RFC 8257, DOI 10.17487/RFC8257, October 2017, <https://www.rfc-editor.org/info/rfc8257>.

[RFC8311] Black, D., "Relaxing Restrictions on Explicit Congestion Notification (ECN) Experimentation", RFC 8311, DOI 10.17487/RFC8311, January 2018, <https://www.rfc-editor.org/info/rfc8311>.

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:

```
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

```
ECEB = r.ceb % 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 3.2.5 and Section 3.2.7); 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:

```
      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

```
      ACE = r.cep % 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 ((newlyAckedB > 0) || (newlyAckedB == 0 && newlyAckedT > 0))
    d.cep = (ACE + DIVACE - (s.cep % DIVACE)) % DIVACE
```

Section 3.2.5 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 AccECN information, because AccECN requires retransmissions to carry the latest AccECN 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 newlyAckedB is the amount of data that an ACK acknowledges since the previous ACK, then the Data Sender could assume that this acknowledges newlyAckedPkt full-sized segments, where newlyAckedPkt = newlyAckedB/MSS. Then it could assume that the ACE field incremented by

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

For example, imagine an ACK acknowledges newlyAckedPkt=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, newlyAckedPkt in the above formula could be replaced with newlyAckedPktHeur = newlyAckedPkt*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 3.2.5 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 AccECN Option

When the AccECN 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 AccECN 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:

```
         ^
   sSafer|
         |
      MSS+
         |
         |         dSafer.cep
         |                  is
    MSS/2+--------------+    safest
         |              |
         | d.cep is safe|
         |    enough    |
         +-------------------->
                       MSS   s
```

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 ~= flightsize / 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 3.2.8 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()
```

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. In this case few options will be sent (evtl. even less than 3 per RTT). However, when continuously sending data, data packets as well as ACKs will spread out equally over the RTT and sufficient ACKs with the AccECN option will be sent.

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.e1b. Note that, because r.e0b is initialized to 1 and the other two counters are initialized 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.

Authors' Addresses

Bob Briscoe
CableLabs
UK

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

Mirja Kuehlewind
ETH Zurich
Zurich
Switzerland

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

Richard Scheffenegger
Vienna
Austria

EMail: rscheff@gmx.at

TCP Maintenance & Minor Extensions (tcpm)
Internet-Draft
Updates: 3168, 3449 (if approved)
Intended status: Standards Track
Expires: May 6, 2021
B. Briscoe
Independent
M. Kuehlewind
Ericsson
R. Scheffenegger
NetApp
November 2, 2020

More Accurate ECN Feedback in TCP
draft-ietf-tcpm-accurate-ecn-13

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). Recent new TCP mechanisms like Congestion Exposure (ConEx), Data Center TCP (DCTCP) or Low Latency Low Loss Scalable Throughput (L4S) need more accurate ECN feedback information whenever more than one marking is received in one RTT. This document specifies a scheme to provide more than one feedback signal per RTT in the TCP header. Given TCP header space is scarce, it allocates a reserved header bit, that was previously used for the ECN-Nonce which has now been declared historic. It also overloads the two existing ECN flags in the TCP header. The resulting extra space is exploited to feed back the IP-ECN field received during the 3-way handshake as well. Supplementary feedback information can optionally be provided in a new TCP option, which is never used on the TCP SYN.

Status of This Memo

This Internet-Draft will expire on May 6, 2021.

Table of Contents

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. In RFC 3168, ECN was specified for TCP in such a way that only one feedback signal could be transmitted per Round-Trip Time (RTT). Recently, proposed mechanisms like Congestion Exposure (ConEx [RFC7713]), DCTCP [RFC8257] or L4S [I-D.ietf-tsvwg-l4s-arch] need to know when more than one marking is received in one RTT which is information that cannot be provided by the feedback scheme as specified in [RFC3168]. This document specifies an update to the ECN feedback scheme of RFC 3168 that provides more accurate information and could be used by these and potentially other future TCP extensions. A fuller treatment of the motivation for this specification is given in the associated requirements document [RFC7560].

This documents specifies a standards track 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 AccECN for short. This document updates RFC 3168 with respect to negotiation and use of the feedback scheme for TCP. All aspects of RFC 3168 other than the TCP feedback scheme, in particular the definition of ECN at the IP layer, remain unchanged by this specification. Section 4 gives a more detailed specification of exactly which aspects of RFC 3168 this document updates.

AccECN is intended to be a complete replacement for classic TCP/ECN feedback, not a fork in the design of TCP. AccECN feedback complements TCP's loss feedback and it can coexist alongside 'classic' [RFC3168] TCP/ECN feedback. So its applicability is intended to include all public and private IP networks (and even any non-IP networks over which TCP is used today), whether or not any nodes on the path support ECN, of whatever flavour. This document uses the term Classic ECN when it needs to distinguish the RFC 3168 ECN TCP feedback scheme from the AccECN TCP feedback scheme.

AccECN feedback overloads the two existing ECN flags in the TCP header and allocates the currently reserved flag (previously called NS) in the TCP header, to be used as one three-bit counter field indicating the number of congestion experienced marked packets. Given the new definitions of these three bits, both ends have to support the new wire protocol before it can be used. Therefore during the TCP handshake the two ends use these three bits in the TCP header to negotiate the most advanced feedback protocol that they can both support, in a way that is backward compatible with [RFC3168].

AccECN is solely a change to the TCP wire protocol; it covers the negotiation and signaling of more accurate ECN feedback from a TCP Data Receiver to a Data Sender. It is completely independent of how TCP might respond to congestion feedback, which is out of scope, but ultimately the motivation for accurate ECN feedback. Like Classic ECN feedback, AccECN can be used by standard Reno congestion control [RFC5681] to respond to the existence of at least one congestion notification within a round trip. Or, unlike Reno, AccECN can be used to respond to the extent of congestion notification over a round trip, as for example DCTCP does in controlled environments [RFC8257]. For congestion response, this specification refers to RFC 3168, or ECN experiments such as those referred to in [RFC8311], namely: a TCP-based Low Latency Low Loss Scalable (L4S) congestion control [I-D.ietf-tsvwg-l4s-arch]; or Alternative Backoff with ECN (ABE) [RFC8511].

It is recommended that the AccECN protocol is implemented alongside SACK [RFC2018] and the experimental ECN++ protocol

[I-D.ietf-tcpm-generalized-ecn], which allows the ECN capability to be used on TCP control packets. Therefore, this specification does not discuss implementing AccECN alongside [RFC5562], which was an earlier experimental protocol with narrower scope than ECN++.

1.1. Document Roadmap

The following introductory section outlines the goals of AccECN (Section 1.2). Then terminology is defined (Section 1.3) and a recap of existing prerequisite technology is given (Section 1.4).

Section 2 gives an informative overview of the AccECN protocol. Then Section 3 gives the normative protocol specification, and Section 4 clarifies which aspects of RFC 3168 are updated by this specification. Section 5 assesses the interaction of AccECN with commonly used variants of TCP, whether standardized or not. Section 6 summarizes the features and properties of AccECN.

Section 7 summarizes the protocol fields and numbers that IANA will need to assign and Section 8 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 AccECN uses and Appendix B explains why AccECN uses flags in the main TCP header and quantifies the space left for future use.

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 recognizes that a perfect scheme that fully satisfies all the requirements is unlikely and trade-offs between requirements are likely. Section 6 presents the properties of AccECN against these requirements and discusses the trade-offs made.

The requirements document recognizes 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. 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 (ACK=1).

Pure ACK: A TCP acknowledgement without a data payload.

Acceptable packet / segment: A packet or segment that passes the acceptability tests in [RFC0793] and [RFC5961].

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 BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.4. 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	Codepoint name	Description
0b00	Not-ECT	Not ECN-Capable Transport
0b01	ECT(1)	ECN-Capable Transport (1)
0b10	ECT(0)	ECN-Capable Transport (0)
0b11	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 last bit in byte 13 of the TCP header was defined as the Nonce Sum (NS) for the ECN Nonce [RFC3540]. In the absence of widespread deployment RFC 3540 has been reclassified as historic [RFC8311] and the respective flag has been marked as "reserved", making this TCP flag available for use by the AccECN experiment instead.

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

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

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. 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 initialized 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 (see Figure 3 later). The 24 LSBs of each byte counter 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. There is no need to acknowledge these continually repeated counters, so the congestion window reduced (CWR) mechanism is no longer used. 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 not allowed 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 [RFC8311]. Because the 3-bit ACE field is so small, when it is the only field available the Data Sender has to interpret it assuming the most likely wrap, but with a degree of conservatism.

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 options are reaching the sender and as long as there is no ACK loss). Implementations are encouraged to send an AccECN 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 AccECN 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 L4S, 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, AccECN 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 Section 3.2.2.3, Section 3.2.2.4 and Section 3.2.3.2 of the present document and para 2 of Section 20.2 of [RFC3168]).

The initial SYN is the most critical control packet, so AccECN provides feedback on its ECN marking. Although RFC 3168 prohibits an ECN-capable SYN, providing feedback of ECN marking on the SYN supports future scenarios in which SYNs might be ECN-enabled (without prejudging whether they ought to be). For instance, [RFC8311] updates this aspect of RFC 3168 to allow experimentation with ECN-capable TCP control packets.

Even if the TCP client (or server) has set the SYN (or SYN/ACK) to not-ECT in compliance with RFC 3168, feedback on the state of the ECN field when it arrives at the receiver could still be useful, because middleboxes have been known to overwrite the ECN IP field as if it is still part of the old Type of Service (ToS) field [Mandalari18]. If a TCP client has set the SYN to Not-ECT, but receives feedback that the ECN field on the SYN arrived with a different codepoint, it can detect such middlebox interference and send Not-ECT for the rest of the connection. Today, if a TCP server receives ECT or 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, prior to AccECN, the server's only safe course of action was to disable ECN for the connection. Instead, the AccECN protocol allows the server to feed back the received ECN field to the client, which then has all the information to decide whether the connection has to fall-back from supporting ECN (or not).

3. AccECN Protocol Specification

3.1. Negotiating to use AccECN

3.1.1. Negotiation during the TCP handshake

Given the ECN Nonce [RFC3540] has been reclassified as historic [RFC8311], the present specification re-allocates the TCP flag at bit 7 of the TCP header, which was previously called NS (Nonce Sum), as the AE (Accurate ECN) flag (see IANA Considerations in Section 7) as shown below.

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

Figure 2: The (post-AccECN) definition of the TCP header flags 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 AE=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 TCP flags on the SYN/ACK to one of the 4 values shown in the top block of Table 2 to confirm that it supports AccECN. The TCP server MUST NOT set one of these 4 combination of flags on the SYN/ACK unless the preceding SYN requested support for AccECN as above.

A TCP server in AccECN mode MUST set the AE, CWR and ECE TCP flags on the SYN/ACK to the value in Table 2 that feeds back the IP-ECN field that arrived on the SYN. This applies whether or not the server itself supports setting the IP-ECN field on a SYN or SYN/ACK (see Section 2.5 for rationale).

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.

Once in AccECN mode, a TCP client or server has the rights and obligations to participate in the ECN protocol defined in Section 3.1.5.

The procedure for the client to follow if a SYN/ACK does not arrive before its retransmission timer expires is given in Section 3.1.4.

3.1.2. Backward Compatibility

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, as above. Table 2 tabulates all the negotiation possibilities for ECN-related capabilities that involve at least one AccECN-capable host. The entries in the first two columns have been abbreviated, as follows:

AccECN: More Accurate ECN Feedback (the present specification)

Nonce: ECN Nonce feedback [RFC3540]

ECN: 'Classic' ECN feedback [RFC3168]

No ECN: Not-ECN-capable. Implicit congestion notification using packet drop.

A	B	SYN A->B			SYN/ACK B->A			Feedback Mode
		AE	CWR	ECE	AE	CWR	ECE	
AccECN	AccECN	1	1	1	0	1	0	AccECN (no ECT on SYN)
AccECN	AccECN	1	1	1	0	1	1	AccECN (ECT1 on SYN)
AccECN	AccECN	1	1	1	1	0	0	AccECN (ECT0 on SYN)
AccECN	AccECN	1	1	1	1	1	0	AccECN (CE on SYN)
AccECN	Nonce	1	1	1	1	0	1	(Reserved)
AccECN	ECN	1	1	1	0	0	1	classic ECN
AccECN	No ECN	1	1	1	0	0	0	Not ECN
Nonce	AccECN	0	1	1	0	0	1	classic ECN
ECN	AccECN	0	1	1	0	0	1	classic ECN
No ECN	AccECN	0	0	0	0	0	0	Not ECN
AccECN	Broken	1	1	1	1	1	1	Not ECN

Table 2: ECN capability negotiation between Client (A) and Server (B)

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

1. The top block shows the case already described in Section 3.1 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. If it has set itself into classic ECN feedback mode it MUST then comply with [RFC3168].

The server response called 'Nonce' in the table is now historic. For an AccECN implementation, there is no need to recognize or support ECN Nonce feedback [RFC3540], which has been reclassified as historic [RFC8311]. AccECN is compatible with alternative ECN feedback integrity approaches (see Section 5.3).

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.

   When an AccECN-enabled TCP server (B) receives a SYN with AE,CWR,ECE = 0,1,1 it MUST do one of the following:

   * set both its half connections into the classic ECN feedback mode and return a SYN/ACK with AE, CWR, ECE = 0,0,1 as shown. Then it MUST comply with [RFC3168].

   * set both its half-connections into No ECN mode and return a SYN/ACK with AE,CWR,ECE = 0,0,0, then continue with ECN disabled. This latter case is unlikely to be desirable, but it is allowed as a possibility, e.g. for minimal TCP implementations.

   When an AccECN-enabled TCP server (B) receives a SYN with AE,CWR,ECE = 0,0,0 it MUST set both its half connections into the Not ECN feedback mode, return a SYN/ACK with AE,CWR,ECE = 0,0,0 as shown and continue with ECN disabled.

4. The fourth block displays a combination labelled 'Broken'. Some older TCP server implementations incorrectly set the reserved flags in the SYN/ACK by reflecting those in the SYN. Such broken TCP servers (B) cannot support ECN, so as soon as an AccECN-capable TCP client (A) receives such a broken SYN/ACK it MUST fall back to Not ECN mode for both its half connections and continue with ECN disabled.

The following additional rules do not fit the structure of the table, but they complement it:

Simultaneous Open: An originating AccECN Host (A), having sent a SYN with AE=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.

In-window SYN during TIME-WAIT: Many TCP implementations create a new TCP connection if they receive an in-window SYN packet during

TIME-WAIT state. When a TCP host enters TIME-WAIT or CLOSED state, it should ignore any previous state about the negotiation of AccECN for that connection and renegotiate the feedback mode according to Table 2.

3.1.3. Forward Compatibility

If a TCP server that implements AccECN receives a SYN with the three TCP header flags (AE, CWR and ECE) set to any combination other than 000, 011 or 111, it MUST negotiate the use of AccECN as if they had been set to 111. This ensures that future uses of the other combinations on a SYN can rely on consistent behaviour from the installed base of AccECN servers.

For the avoidance of doubt, the behaviour described in the present specification applies whether or not the three remaining reserved TCP header flags are zero.

3.1.4. Retransmission of the SYN

If the sender of an AccECN SYN times out before receiving the SYN/ACK, the sender SHOULD attempt to negotiate the use of AccECN at least one more time by continuing to set all three TCP ECN flags on the first retransmitted SYN (using the usual retransmission time-outs). If this first retransmission also fails to be acknowledged, the sender SHOULD send subsequent retransmissions of the SYN with the three TCP-ECN flags cleared (AE=CWR=ECE=0). A retransmitted SYN MUST use the same ISN as the original SYN.

Retrying once before fall-back adds delay in the case where a middlebox drops an AccECN (or ECN) SYN deliberately. However, current measurements imply that a drop is less likely to be due to middlebox interference than other intermittent causes of loss, e.g. congestion, wireless interference, etc.

Implementers MAY use other fall-back strategies if they are found to be more effective (e.g. attempting to negotiate AccECN on the SYN only once or more than twice (most appropriate during high levels of congestion). However, other fall-back strategies will need to follow all the rules in Section 3.1.5, which concern behaviour when SYNs or SYN/ACKs negotiating different types of feedback have been sent within the same connection.

Further it may make sense to also remove any other new or 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.

Whichever fall-back strategy is used, the TCP initiator SHOULD cache failed connection attempts. If it does, it SHOULD NOT give up attempting to negotiate AccECN on the SYN of subsequent connection attempts until it is clear that the blockage is persistently and specifically due to AccECN. The cache should be arranged to expire so that the initiator will infrequently attempt to check whether the problem has been resolved.

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 3.2.3.2.

3.1.5. Implications of AccECN Mode

Section 3.1.1 describes the only ways that a host can enter AccECN mode, whether as a client or as a server.

As a Data Sender, a host in AccECN mode has the rights and obligations concerning the use of ECN defined below, which build on those in [RFC3168] as updated by [RFC8311]:

- Using ECT:

  * It can set an ECT codepoint in the IP header of packets to indicate to the network that the transport is capable and willing to participate in ECN for this packet.

  * It does not have to set ECT on any packet (for instance if it has reason to believe such a packet would be blocked).

- Switching feedback negotiation (e.g. fall-back):

  * It SHOULD NOT set ECT on any packet if it has received at least one valid SYN or Acceptable SYN/ACK with AE=CWR=ECE=0. A "valid SYN" has the same port numbers and the same ISN as the SYN that caused the server to enter AccECN mode.

  * It MUST NOT send an ECN-setup SYN [RFC3168] within the same connection as it has sent a SYN requesting AccECN feedback.

  * It MUST NOT send an ECN-setup SYN/ACK [RFC3168] within the same connection as it has sent a SYN/ACK agreeing to use AccECN feedback.

  The above rules are necessary because, when one peer negotiates the feedback mode in two different types of handshake, it is not possible for the other peer to know for certain which handshake packet(s) the other end eventually receives or in which order it receives them. So the two peers can end up using difference feedback modes without knowing it.

- Congestion response:

  * It is still obliged to respond appropriately to AccECN feedback with congestion indications on packets it had previously sent, as defined in Section 6.1 of [RFC3168] and updated by Sections 2.1 and 4.1 of [RFC8311].

  * The commitment to respond appropriately to incoming indications of congestion remains even if it sends a SYN packet with AE=CWR=ECE=0, in a later transmission within the same TCP connection.

  * Unlike an RFC 3168 data sender, it MUST NOT set CWR to indicate it has received and responded to indications of congestion (for the avoidance of doubt, this does not preclude it from setting the bits of the ACE counter field, which includes an overloaded use of the same bit).

As a Data Receiver:

- a host in AccECN mode MUST feed back the information in the IP-ECN field on incoming packets using Accurate ECN feedback, as specified in Section 3.2 below.

- if it receives an ECN-setup SYN or ECN-setup SYN/ACK [RFC3168] during the same connection as it receives a SYN requesting AccECN feedback or a SYN/ACK agreeing to use AccECN feedback, it MUST reset the connection with a RST packet.

- If for any reason it is not willing to provide ECN feedback on a particular TCP connection, to indicate this unwillingness it SHOULD clear the AE, CWR and ECE flags in all SYN and/or SYN/ACK packets that it sends.

- it MUST NOT use reception of packets with ECT set in the IP-ECN field as an implicit signal that the peer is ECN-capable. Reason: ECT at the IP layer does not explicitly confirm the peer has the correct ECN feedback logic, and the packets could have been mangled at the IP layer.

3.2. AccECN Feedback

Each Data Receiver of each half connection maintains four counters, r.cep, r.ceb, r.e0b and r.e1b:

- o The Data Receiver MUST increment the CE packet counter (r.cep), for every Acceptable packet that it receives with the CE code point in the IP ECN field, including CE marked control packets but excluding CE on SYN packets (SYN=1; ACK=0).

- o The Data Receiver MUST increment the r.ceb, r.e0b or r.e1b byte counters by the number of TCP payload octets in Acceptable packets marked respectively with the CE, ECT(0) and ECT(1) codepoint in their IP-ECN field, including any payload octets on control packets, but not including any payload octets on SYN packets (SYN=1; ACK=0).

Each Data Sender of each half connection maintains four counters, s.cep, s.ceb, s.e0b and s.e1b intended to track the equivalent counters at the Data Receiver.

A Data Receiver feeds back the CE packet counter using the Accurate ECN (ACE) field, as explained in Section 3.2.2. 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. Therefore the feedback carried on a retransmitted packet is unlikely to be the same as the feedback on the original packet.

3.2.1. Initialization of Feedback Counters

When a host first enters AccECN mode, in its role as a Data Receiver it initializes its counters to r.cep = 5, r.e0b = 1 and r.ceb = r.e1b.= 0,

Non-zero initial values are used to support a stateless handshake (see Section 5.1) and to be distinct from cases where the fields are incorrectly zeroed (e.g. by middleboxes - see Section 3.2.3.2.4).

When a host enters AccECN mode, in its role as a Data Sender it initializes its counters to s.cep = 5, s.e0b = 1 and s.ceb = s.e1b.= 0.

3.2.2. The ACE Field

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

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

Figure 3: 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 to ACE unconditionally; it merely overloads them with another name and definition once an AccECN connection has been established.

With one exception (Section 3.2.2.1), a host with both of its half-connections in AccECN mode MUST interpret the AE, CWR and ECE flags as the 3-bit ACE counter on a segment with the SYN flag cleared (SYN=0). On such a packet, a Data Receiver MUST encode the three least significant bits of its r.cep counter into the ACE field that it feeds back to the Data Sender. 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).

3.2.2.1. ACE Field on the ACK of the SYN/ACK

A TCP client (A) in AccECN mode MUST feed back which of the 4 possible values of the IP-ECN field was on the SYN/ACK by writing it into the ACE field of a pure ACK with no SACK blocks using the binary encoding in Table 3 (which is the same as that used on the SYN/ACK in Table 2). This shall be called the handshake encoding of the ACE field, and it is the only exception to the rule that the ACE field carries the 3 least significant bits of the r.cep counter on packets with SYN=0.

Normally, a TCP client acknowledges a SYN/ACK with an ACK that satisfies the above conditions anyway (SYN=0, no data, no SACK blocks). If an AccECN TCP client intends to acknowledge the SYN/ACK with a packet that does not satisfy these conditions (e.g. it has data to include on the ACK), it SHOULD first send a pure ACK that does satisfy these conditions (see Section 5.2), so that it can feed back which of the four values of the IP-ECN field arrived on the SYN/ACK. A valid exception to this "SHOULD" would be where the implementation will only be used in an environment where mangling of the ECN field is unlikely.

IP-ECN codepoint on SYN/ACK	ACE on pure ACK of SYN/ACK	r.cep of client in AccECN mode
Not-ECT	0b010	5
ECT(1)	0b011	5
ECT(0)	0b100	5
CE	0b110	6

Table 3: The encoding of the ACE field in the ACK of the SYN-ACK to reflect the SYN-ACK's IP-ECN field

When an AccECN server in SYN-RCVD state receives a pure ACK with SYN=0 and no SACK blocks, instead of treating the ACE field as a counter, it MUST infer the meaning of each possible value of the ACE field from Table 4, which also shows the value that an AccECN server MUST set s.cep to as a result.

Given this encoding of the ACE field on the ACK of a SYN/ACK is exceptional, an AccECN server using large receive offload (LRO) might prefer to disable LRO until such an ACK has transitioned it out of SYN-RCVD state.

ACE on ACK of SYN/ACK	IP-ECN codepoint on SYN/ACK inferred by server	s.cep of server in AccECN mode
0b000	{Notes 1, 3}	Disable ECN
0b001	{Notes 2, 3}	5
0b010	Not-ECT	5
0b011	ECT(1)	5
0b100	ECT(0)	5
0b101	Currently Unused {Note 2}	5
0b110	CE	6
0b111	Currently Unused {Note 2}	5

Table 4: Meaning of the ACE field on the ACK of the SYN/ACK

{Note 1}: If the server is in AccECN mode, the value of zero raises suspicion of zeroing of the ACE field on the path (see Section 3.2.2.3).

{Note 2}: If the server is in AccECN mode, these values are Currently Unused but the AccECN server's behaviour is still defined for forward compatibility. Then the designer of a future protocol can know for certain what AccECN servers will do with these codepoints.

{Note 3}: In the case where a server that implements AccECN is also using a stateless handshake (termed a SYN cookie) it will not remember whether it entered AccECN mode. The values 0b000 or 0b001 will remind it that it did not enter AccECN mode, because AccECN does not use them (see Section 5.1 for details). If a stateless server that implements AccECN receives either of these two values in the ACK, its action is implementation-dependent and outside the scope of this spec, It will certainly not take the action in the third column because, after it receives either of these values, it is not in AccECN mode. I.e., it will not disable ECN (at least not just because ACE is 0b000) and it will not set s.cep.

3.2.2.2. Encoding and Decoding Feedback in the ACE Field

Whenever the Data Receiver sends an ACK with SYN=0 (with or without data), unless the handshake encoding in Section 3.2.2.1 applies, the Data Receiver MUST encode the least significant 3 bits of its r.cep counter into the ACE field (see Appendix A.2).

Whenever the Data Sender receives an ACK with SYN=0 (with or without data), it first checks whether it has already been superseded by another ACK in which case it ignores the ECN feedback. If the ACK has not been superseded, and if the special handshake encoding in Section 3.2.2.1 does not apply, the Data Sender decodes the ACE field as follows (see Appendix A.2 for examples).

o It takes the least significant 3 bits of its local s.cep counter and subtracts them from the incoming ACE counter to work out the minimum positive increment it could apply to s.cep (assuming the ACE field only wrapped at most once).

o It then follows the safety procedures in Section 3.2.2.5.2 to calculate or estimate how many packets the ACK could have acknowledged under the prevailing conditions to determine whether the ACE field might have wrapped more than once.

The encode/decode procedures during the three-way handshake are exceptions to the general rules given so far, so they are spelled out step by step below for clarity:

o If a TCP server in AccECN mode receives a CE mark in the IP-ECN field of a SYN (SYN=1, ACK=0), it MUST NOT increment r.cep (it remains at its initial value of 5).

Reason: It would be redundant for the server to include CE-marked SYNs in its r.cep counter, because it already reliably delivers feedback of any CE marking on the SYN/ACK using the encoding in Table 2. This also ensures that, when the server starts using the ACE field, it has not unnecessarily consumed more than one initial value, given they can be used to negotiate variants of the AccECN protocol (see Appendix B.3).

o If a TCP client in AccECN mode receives CE feedback in the TCP flags of a SYN/ACK, it MUST NOT increment s.cep (it remains at its initial value of 5), so that it stays in step with r.cep on the server. Nonetheless, the TCP client still triggers the congestion control actions necessary to respond to the CE feedback.

o If a TCP client in AccECN mode receives a CE mark in the IP-ECN field of a SYN/ACK, it MUST increment r.cep, but no more than once no matter how many CE-marked SYN/ACKs it receives (i.e. incremented from 5 to 6, but no further).

Reason: Incrementing r.cep ensures the client will eventually deliver any CE marking to the server reliably when it starts using the ACE field. Even though the client also feeds back any CE marking on the ACK of the SYN/ACK using the encoding in Table 3, this ACK is not delivered reliably, so it can be considered as a timely notification that is redundant but unreliable. The client does not increment r.cep more than once, because the server can only increment s.cep once (see next bullet). Also, this limits the unnecessarily consumed initial values of the ACE field to two.

o If a TCP server in AccECN mode and in SYN-RCVD state receives CE feedback in the TCP flags of a pure ACK with no SACK blocks, it MUST increment s.cep (from 5 to 6). The TCP server then triggers the congestion control actions necessary to respond to the CE feedback.

Reasoning: The TCP server can only increment s.cep once, because the first ACK it receives will cause it to transition out of SYN-RCVD state. The server's congestion response would be no different even if it could receive feedback of more than one CE-marked SYN/ACK.

Once the TCP server transitions to ESTABLISHED state, it might later receive other pure ACK(s) with the handshake encoding in the ACE field. The conditions for this to occur are quite unusual, but not impossible, e.g. a SYN/ACK (or ACK of the SYN/ACK) that is delayed for longer than the server's retransmission timeout; or packet duplication by the network. Nonetheless, once in the ESTABLISHED state, the server will consider the ACE field to be encoded as the normal ACE counter on all packets with SYN=0 (given it will be following the above rule in this bullet). The server MAY include a test to avoid this case.

3.2.2.3. Testing for Zeroing of the ACE Field

Section 3.2.2 required the Data Receiver to initialize the r.cep counter to a non-zero value. Therefore, in either direction the initial value of the ACE counter ought to be non-zero.

If AccECN has been successfully negotiated, the Data Sender SHOULD check the value of the ACE counter in the first packet (with or without data) that arrives with SYN=0. If the value of this ACE field is zero (0b000), the Data Sender disables sending ECN-capable packets for the remainder of the half-connection by setting the IP/ECN field in all subsequent packets to Not-ECT.

Usually, the server checks the ACK of the SYN/ACK from the client, while the client checks the first data segment from the server. However, if reordering occurs, "the first packet ... that arrives" will not necessarily be the same as the first packet in sequence order. The test has been specified loosely like this to simplify implementation, and because it would not have been any more precise to have specified the first packet in sequence order, which would not necessarily be the first ACE counter that the Data Receiver fed back anyway, given it might have been a retransmission.

The possibility of re-ordering means that there is a small chance that the ACE field on the first packet to arrive is genuinely zero (without middlebox interference). This would cause a host to unnecessarily disable ECN for a half connection. Therefore, in environments where there is no evidence of the ACE field being zeroed, implementations can skip this test.

Note that the Data Sender MUST NOT test whether the arriving counter in the initial ACE field has been initialized to a specific valid value - the above check solely tests whether the ACE fields have been incorrectly zeroed. This allows hosts to use different initial values as an additional signalling channel in future.

3.2.2.4. Testing for Mangling of the IP/ECN Field

The value of the ACE field on the SYN/ACK indicates the value of the IP/ECN field when the SYN arrived at the server. The client can compare this with how it originally set the IP/ECN field on the SYN. If this comparison implies an unsafe transition (see below) of the IP/ECN field, for the remainder of the connection the client MUST NOT send ECN-capable packets, but it MUST continue to feed back any ECN markings on arriving packets.

The value of the ACE field on the last ACK of the 3WHS indicates the value of the IP/ECN field when the SYN/ACK arrived at the client. The server can compare this with how it originally set the IP/ECN field on the SYN/ACK. If this comparison implies an unsafe transition of the IP/ECN field, for the remainder of the connection the server MUST NOT send ECN-capable packets, but it MUST continue to feed back any ECN markings on arriving packets.

The ACK of the SYN/ACK is not reliably delivered (nonetheless, the count of CE marks is still eventually delivered reliably). If this ACK does not arrive, the server can continue to send ECN-capable packets without having tested for mangling of the IP/ECN field on the SYN/ACK.

Invalid transitions of the IP/ECN field are defined in [RFC3168] and repeated here for convenience:

- the not-ECT codepoint changes;

- either ECT codepoint transitions to not-ECT;

- the CE codepoint changes.

RFC 3168 says that a router that changes ECT to not-ECT is invalid but safe. However, from a host's viewpoint, this transition is unsafe because it could be the result of two transitions at different routers on the path: ECT to CE (safe) then CE to not-ECT (unsafe). This scenario could well happen where an ECN-enabled home router congests its upstream mobile broadband bottleneck link, then the ingress to the mobile network clears the ECN field [Mandalari18].

Once a Data Sender has entered AccECN mode it SHOULD check whether all feedback received for the first three or four round indicated that every packet it sent was CE-marked. If so, for the remainder of the connection, the Data Sender SHOULD NOT send ECN-capable packets, but it MUST continue to feed back any ECN markings on arriving packets.

The above fall-back behaviours are necessary in case mangling of the IP/ECN field is asymmetric, which is currently common over some mobile networks [Mandalari18]. Then one end might see no unsafe transition and continue sending ECN-capable packets, while the other end sees an unsafe transition and stops sending ECN-capable packets.

3.2.2.5. 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 or thinned out, the 3-bit counter in the ACE field might have cycled between two ACKs arriving at the Data Sender. The following safety procedures minimize this ambiguity.

3.2.2.5.1. Data Receiver Safety Procedures

An AccECN Data Receiver:

- SHOULD immediately send an ACK whenever a data packet marked CE arrives after the previous data packet was not CE.

- MUST 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.

These rules for when to send an ACK are designed to be complemented by those in Section 3.2.3.3, which concern whether the AccECN TCP Option ought to be included on ACKs.

For the avoidance of doubt, the change-triggered ACK mechanism is deliberately worded to solely apply to data packets, and to ignore the arrival of a control packet with no payload, because it is important that TCP does not acknowledge pure ACKs. The change-triggered ACK approach can lead to some additional ACKs but it feeds back the timing and the order in which ECN marks are received with minimal additional complexity. If only CE marks are infrequent, or there are multiple marks in a row, the additional load will be low. Other marking patterns could increase the load significantly.

Even though the first bullet is stated as a "SHOULD", it is important for a transition to immediately trigger an ACK if at all possible, so that the Data Sender can rely on change-triggered ACKs to detect queue growth as soon as possible, e.g. at the start of a flow. This requirement can only be relaxed if certain offload hardware needed for high performance cannot support change-triggered ACKs (although high performance protocols such as DCTCP already successfully use change-triggered ACKs). One possible compromise would be for the receiver to heuristically detect whether the sender is in slow-start, then to implement change-triggered ACKs while the sender is in slow-start, and offload otherwise.

3.2.2.5.2. Data Sender Safety Procedures

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, it SHOULD deem whether it cycled by taking the safest likely case under the prevailing conditions. 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 neutralize the effect of any action it took based on a conservative assumption that it later found to be incorrect.

The Data Sender can estimate how many packets (of any marking) an ACK acknowledges. If the ACE counter on an ACK seems to imply that the minimum number of newly CE-marked packets is greater that the number of newly acknowledged packets, the Data Sender SHOULD believe the ACE counter, unless it can be sure that it is counting all control packets correctly.

3.2.3. The AccECN Option

The AccECN Option is defined as shown in Figure 4. The initial 'E' of each field name stands for 'Echo'.

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |  Kind = TBD0  |  Length = 11  |          EE0B field           |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   | EE0B (cont'd) |           ECEB field                          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                  EE1B field                   |         Order 0
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |  Kind = TBD1  |  Length = 11  |          EE1B field           |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   | EE1B (cont'd) |           ECEB field                          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                  EE0B field                   |         Order 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4: The AccECN TCP Option

Figure 4 shows two option field orders; order 0 and order 1. They both consists of three 24-bit fields. Order 0 provides the 24 least significant bits of the r.e0b, r.ceb and r.e1b counters, respectively. Order 1 provides the same fields, but in the opposite order. On each packet, the Data Receiver can use whichever order is more efficient.

When a Data Receiver sends an AccECN Option, it MUST set the Kind field to TBD0 if using Order 0, or to TBD1 if using Order 1. These two new TCP Option Kinds are registered in Section 7 and called respectively AccECN0 and AccECN1.

Note that there is no field to feed back 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 3.2.3.3 expect it to usually 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	Type 0	Type 1
11	EE0B, ECEB, EE1B	EE1B, ECEB, EE0B
8	EE0B, ECEB	EE1B, ECEB
5	EE0B	EE1B
2	(empty)	(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.

All implementations of a Data Sender that read any AccECN Option MUST be able to read in AccECN Options of any of the above lengths. For forward compatibility, if the AccECN Option is of any other length, implementations MUST use those whole 3-octet fields that fit within the length and ignore the remainder of the option.

The AccECN Option has to be optional to implement, because both sender and receiver have to be able to cope without the option anyway - in cases where it does not traverse a network path. It is RECOMMENDED to implement both sending and receiving of the AccECN Option. If sending of the AccECN Option is implemented, the fall-backs described in this document will need to be implemented as well (unless solely for a controlled environment where path traversal is not considered a problem). Even if a developer does not implement sending of the AccECN Option, it is RECOMMENDED that they still implement logic to receive and understand any AccECN Options sent by remote peers.

If a Data Receiver intends to send the AccECN Option at any time during the rest of the connection it is strongly recommended to also test path traversal of the AccECN Option as specified in Section 3.2.3.2.

3.2.3.1. Encoding and Decoding Feedback in the AccECN Option Fields

Whenever the Data Receiver includes any of the counter fields (ECEB, EE0B, EE1B) in an AccECN Option, it MUST encode the 24 least significant bits of the current value of the associated counter into the field (respectively r.ceb, r.e0b, r.e1b).

Whenever the Data Sender receives ACK carrying an AccECN Option, it first checks whether the ACK has already been superseded by another ACK in which case it ignores the ECN feedback. If the ACK has not been superseded, the Data Sender MUST decode the fields in the AccECN Option as follows. For each field, it takes the least significant 24 bits of its associated local counter (s.ceb, s.e0b or s.e1b) and subtracts them from the counter in the associated field of the incoming AccECN Option (respectively ECEB, EE0B, EE1B), to work out the minimum positive increment it could apply to s.ceb, s.e0b or s.e1b (assuming the field in the option only wrapped at most once).

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, as specified in Section 3.2, any data on the SYN (SYN=1, ACK=0) is not included in any of the locally held octet counters nor in the AccECN Option on the wire.

3.2.3.2. Path Traversal of the AccECN Option

3.2.3.2.1. Testing the AccECN Option during the Handshake

The TCP client MUST NOT include the AccECN TCP Option on the SYN. (A fall-back strategy for the loss of the SYN (possibly due to middlebox interference) is specified in Section 3.1.4.)

A TCP server that confirms its support for AccECN (in response to an AccECN SYN from the client as described in Section 3.1) SHOULD include an AccECN TCP Option on 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 3WHS. 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 MAY 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.

3.2.3.2.2. Testing for Loss of Packets Carrying the AccECN Option

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 TCP server SHOULD retransmit the SYN/ACK repeating the same AE, CWR and ECE TCP flags as on the original SYN/ACK but with no AccECN Option. If this retransmission times out, to expedite connection setup, the TCP server SHOULD disable AccECN and ECN for this connection by retransmitting the SYN/ACK with AE=CWR=ECE=0 and no AccECN Option.

Implementers MAY use other fall-back strategies if they are found to be more effective (e.g. retrying the AccECN Option for a second time before fall-back - most appropriate during high levels of congestion). However, other fall-back strategies will need to follow all the rules in Section 3.1.5, which concern behaviour when SYNs or SYN/ACKs negotiating different types of feedback have been sent within the same connection.

If the TCP client detects that the first data segment it sent with the AccECN Option 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 whether the AccECN Option is blocked for subsequent connections. [I-D.ietf-tcpm-2140bis] further discusses caching of TCP parameters and status information.

If a host falls back to not sending the AccECN Option, it will continue to process any incoming AccECN Options as normal.

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

If the TCP server receives a second SYN with a request for AccECN support, it should resend the SYN/ACK, again confirming its support for AccECN, but this time without the AccECN Option. This approach rules out any interference by middleboxes that may drop packets with unknown options, even though it is more likely that the SYN/ACK would have been lost due to congestion. The TCP server MAY try to send another packet with the AccECN Option at a later point during the connection but should monitor if that packet got lost as well, in which case it SHOULD disable the sending of the AccECN Option for this half-connection.

Similarly, an AccECN end-point MAY separately memorize which data packets carried an AccECN Option and disable the sending of AccECN Options if the loss probability of those packets is significantly higher than that of all other data packets in the same connection.

3.2.3.2.3. Testing for Absence of the AccECN Option

If the TCP client has successfully negotiated AccECN but does not receive an AccECN Option on the SYN/ACK (e.g. because is has been stripped by a middlebox or not sent by the server), the client 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 segment that acknowledges sequence space at least covering the ISN, it switches into a mode that assumes that the AccECN Option is not available for this half connection.

While a host is in this mode that assumes incoming AccECN Options are not available, it MUST adopt the conservative interpretation of the ACE field discussed in Section 3.2.2.5. However, it cannot make any assumption about support of outgoing AccECN Options on the other half connection, so it SHOULD continue to send the AccECN Option itself (unless it has established that sending the AccECN Option is causing packets to be blocked as in Section 3.2.3.2.2).

If a host is in the mode that assumes incoming AccECN Options are not available, but it receives an AccECN Option at any later point during the connection, this clearly indicates that the AccECN Option is not blocked on the respective path, and the AccECN endpoint MAY switch out of the mode that assumes the AccECN Option is not available for this half connection.

3.2.3.2.4. Test for Zeroing of the AccECN Option

For a related test for invalid initialization of the ACE field, see Section 3.2.2.3

Section 3.2 required the Data Receiver to initialize the r.e0b counter to a non-zero value. Therefore, in either direction the initial value of the EE0B field in the AccECN Option (if one exists) ought to be non-zero. If AccECN has been negotiated:

o the TCP server MAY check the initial value of the EE0B field in the first segment that acknowledges sequence space that at least covers the ISN plus 1. If the initial value of the EE0B field is zero, the server will switch into a mode that ignores the AccECN Option for this half connection.

o the TCP client MAY check the initial value of the EE0B field on the SYN/ACK. If the initial value of the EE0B field is zero, the client will switch into a mode that ignores the AccECN Option for this half connection.

While a host is in the mode that ignores the AccECN Option it MUST adopt the conservative interpretation of the ACE field discussed in Section 3.2.2.5.

Note that the Data Sender MUST NOT test whether the arriving byte counters in the initial AccECN Option have been initialized to specific valid values - the above checks solely test whether these fields have been incorrectly zeroed. This allows hosts to use different initial values as an additional signalling channel in future. Also note that the initial value of either field might be greater than its expected initial value, because the counters might already have been incremented. Nonetheless, the initial values of the counters have been chosen so that they cannot wrap to zero on these initial segments.

3.2.3.2.5. Consistency between AccECN Feedback Fields

When the AccECN Option is available it supplements but does not replace the ACE field. An endpoint using AccECN feedback MUST always consider the information provided in the ACE field whether or not the AccECN Option is also available.

If the AccECN option is present, the s.cep counter might increase while the s.ceb counter does not (e.g. due to a CE-marked control packet). The sender's response to such a situation is out of scope, and needs to be dealt with in a specification that uses ECN-capable control packets. Theoretically, this situation could also occur if a middlebox mangled the AccECN Option but not the ACE field. However, the Data Sender has to assume that the integrity of the AccECN Option is sound, based on the above test of the well-known initial values and optionally other integrity tests (Section 5.3).

If either end-point detects that the s.ceb counter has increased but the s.cep has not (and by testing ACK coverage it is certain how much the ACE field has wrapped), this invalid protocol transition has to be due to some form of feedback mangling. So, the Data Sender MUST disable sending ECN-capable packets for the remainder of the half-connection by setting the IP/ECN field in all subsequent packets to Not-ECT.

3.2.3.3. Usage of the AccECN TCP Option

If the Data Receiver intends to use the AccECN TCP Option to provide feedback, the following rules determine when a Data Receiver in AccECN mode sends an ACK with 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 (this is in addition to the safety recommendation in Section 3.2.2.5 against ambiguity of the ACE field).

Even though this bullet is stated as a "SHOULD", it is important for a transition to immediately trigger an ACK if at all possible, as already argued when specifying change-triggered ACKs for the ACE.

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.

* It SHOULD include a counter that has continued to increment on the next scheduled ACK following a change-triggered ACK;

* while the same counter continues to increment, it SHOULD include the counter every n ACKs as consistently as possible, where n can be chosen by the implementer;

* It SHOULD always include an AccECN Option if the r.ceb counter is incrementing and it MAY include an AccECN Option if r.ec0b or r.ec1b is incrementing

* It SHOULD, include each counter at least once for every 2^22 bytes incremented to prevent overflow during continual repetition.

If the smallest allowed AccECN Option would leave insufficient space for two SACK blocks on a particular ACK, the Data Receiver MUST give precedence to the SACK option (total 18 octets), because loss feedback is more critical.

Necessary Option Length: It MAY exclude counter(s) that have not changed for the whole connection (but beacons still include all fields - see below). It SHOULD include counter(s) that have incremented at some time during the connection. 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 above rules complement those in Section 3.2.2.5, which determine when to generate an ACK irrespective of whether an AccECN TCP Option is to be included.

The following example series of arriving IP/ECN fields illustrates when a Data Receiver will emit an ACK with an AccECN Option 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.

Even though first bullet is stated as a "SHOULD", it is important for a transition to immediately trigger an ACK if at all possible, so that the Data Sender can rely on change-triggered ACKs to detect queue growth as soon as possible, e.g. at the start of a flow. This requirement can only be relaxed if certain offload hardware needed for high performance cannot support change-triggered ACKs (although high performance protocols such as DCTCP already successfully use change-triggered ACKs). One possible experimental compromise would be for the receiver to heuristically detect whether the sender is in slow-start, then to implement change-triggered ACKs while the sender is in slow-start, and offload otherwise.

For the avoidance of doubt, this change-triggered ACK mechanism is deliberately worded to ignore the arrival of a control packet with no payload, which therefore does not alter any byte counters, because it is important that TCP does not acknowledge pure ACKs. The change-triggered ACK approach can lead to some additional ACKs but it feeds back the timing and the order in which ECN marks are received with minimal additional complexity. If only CE marks are infrequent, or there are multiple marks in a row, the additional load will be low. Other marking patterns could increase the load significantly, Investigating the additional load is a goal of the proposed experiment.

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 Requirements for TCP Proxies, Offload Engines and other Middleboxes

3.3.1. Requirements for TCP Proxies

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.

3.3.2. Requirements for TCP Normalizers

Another large class of middleboxes intervenes 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 'normalize' the TCP wire protocol by checking that all values in header fields comply with a rather narrow and often outdated 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.

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 standardized set of values being extended in future (because outdated normalizers would block updated hosts from using the extended AccECN standard).

3.3.3. Requirements for TCP ACK Filtering

A node that implements ACK filtering (aka. thinning or coalescing) and itself also implements ECN marking will not need to filter ACKs from connections that use AccECN feedback. Therefore, such a node SHOULD detect connections that have negotiated the use of AccECN feedback during the handshake (see Table 2) and it SHOULD preserve the timing of each ACK (if it coalesced ACKs it would not be AccECN-compliant, but the requirement is stated as a "SHOULD" in order to allow leeway for pre-existing ACK filtering functions to be brought into line).

A node that implements ACK filtering and does not itself implement ECN marking does not need to treat AccECN connections any differently from other TCP connections. Nonetheless, it is RECOMMENDED that such nodes implement ECN marking and comply with the requirements of the previoius paragraph. This should be a better way than ACK filtering to improve the performance of AccECN TCP connections.

The rationale for these requirements is that AccECN feedback provides sufficient information to a data receiver for it to be able to monitor ECN marking of the ACKs it has sent, so that it can thin the ACK stream itself. This will eventually mean that ACK filtering in the network gives no performance advantage. Then TCP will be able to maintain its own control over ACK coalescing. This will also allow the TCP Data Sender to use the timing of ACK arrivals to more reliably infer further information about the path congestion level.

Note that the specification of AccECN in TCP does not presume to rely on the above ACK filtering behaviour in the network, because it has to be robust against pre-existing network nodes that still filter AccECN ACKs, and robust against ACK loss during overload.

Section 5.2.1 of [RFC3449] gives best current practice on ACK filtering (aka. thinning or coalescing). It gives no advice on ACKs carrying ECN feedback, because at the time is said that "ECN remain areas of ongoing research". This section updates that advice for a TCP connection that supports AccECN feedback.

3.3.4. Requirements for TCP Segmentation Offload

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').

The ACE field changes with every received CE marking, so today's receive offloading could lead to many interrupts in high congestion situations. Although that would be useful (because congestion information is received sooner), it could also significantly increase processor load, particularly in scenarios such as DCTCP or L4S where the marking rate is generally higher.

Current offload hardware ejects a segment from the coalescing process whenever the TCP ECN flags change. Thus Classic ECN causes offload to be inefficient. In data centres it has been fortunate for this offload hardware that DCTCP-style feedback changes less often when there are long sequences of CE marks, which is more common with a step marking threshold (but less likely the more short flows are in the mix). The ACE counter approach has been designed so that coalescing can continue over arbitrary patterns of marking and only needs to stop when the counter wraps. Nonetheless, until the particular offload hardware in use implements this more efficient approach, it is likely to be more efficient for AccECN connections to implement this counter-style logic using software segmentation offload.

ECN encodes a varying signal in the ACK stream, so it is inevitable that offload hardware will ultimately need to handle any form of ECN feedback exceptionally. The ACE field has been designed as a counter so that it is straightforward for offload hardware to pass on the highest counter, and to push a segment from its cache before the counter wraps. The purpose of working towards standardized TCP ECN feedback is to reduce the risk for hardware developers, who would otherwise have to guess which scheme is likely to become dominant.

The above process has been designed to enable a continuing incremental deployment path - to more highly dynamic congestion control. Once DCTCP offload hardware supports AccECN, it will be able to coalesce efficiently for any sequence of marks, instead of relying for efficiency on the long marking sequences from step marking. In the next stage, DCTCP marking can evolve from a step to a ramp function. That in turn will allow host congestion control algorithms to respond faster to dynamics, while being backwards compatible with existing host algorithms.

4. Updates to RFC 3168

Normative statements in the following sections of RFC3168 are updated by the present AccECN specification:

- The whole of "6.1.1 TCP Initialization" of [RFC3168] is updated by Section 3.1 of the present specification.

- In "6.1.2. The TCP Sender" of [RFC3168], all mentions of a congestion response to an ECN-Echo (ECE) ACK packet are updated by Section 3.2 of the present specification to mean an increment to the sender's count of CE-marked packets, s.cep. And the requirements to set the CWR flag no longer apply, as specified in Section 3.1.5 of the present specification. Otherwise, the remaining requirements in "6.1.2. The TCP Sender" still stand.

  It will be noted that RFC 8311 already updates, or potentially updates, a number of the requirements in "6.1.2. The TCP Sender". Section 6.1.2 of RFC 3168 extended standard TCP congestion control [RFC5681] to cover ECN marking as well as packet drop. Whereas, RFC 8311 enables experimentation with alternative responses to ECN marking, if specified for instance by an experimental RFC on the IETF document stream. RFC 8311 also strengthened the statement that "ECT(0) SHOULD be used" to a "MUST" (see [RFC8311] for the details).

o The whole of "6.1.3. The TCP Receiver" of [RFC3168] is updated by Section 3.2 of the present specification, with the exception of the last paragraph (about congestion response to drop and ECN in the same round trip), which still stands. Incidentally, this last paragraph is in the wrong section, because it relates to TCP sender behaviour.

o The following text within "6.1.5. Retransmitted TCP packets":

> "the TCP data receiver SHOULD ignore the ECN field on arriving data packets that are outside of the receiver's current window."

is updated by more stringent acceptability tests for any packet (not just data packets) in the present specification. Specifically, in the normative specification of AccECN (Section 3) only 'Acceptable' packets contribute to the ECN counters at the AccECN receiver and Section 1.3 defines an Acceptable packet as one that passes the acceptability tests in both [RFC0793] and [RFC5961].

o Sections 5.2, 6.1.1, 6.1.4, 6.1.5 and 6.1.6 of [RFC3168] prohibit use of ECN on TCP control packets and retransmissions. The present specification does not update that aspect of RFC 3168, but it does say what feedback an AccECN Data Receiver should provide if it receives an ECN-capable control packet or retransmission. This ensures AccECN is forward compatible with any future scheme that allows ECN on these packets, as provided for in section 4.3 of [RFC8311] and as proposed in [I-D.ietf-tcpm-generalized-ecn].

5. Interaction with TCP Variants

This section is informative, not normative.

5.1. 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 it receives a pure ACK that acknowledges an ISN that is a valid SYN cookie, and if the ACK contains an ACE field with the value 0b010 to 0b111 (decimal 2 to 7), 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.

If the pure ACK that acknowledges a SYN cookie contains an ACE field with the value 0b000 or 0b001, these values indicate that the client did not request support for AccECN and therefore the server does not enter AccECN mode for this connection. Further, 0b001 on the ACK implies that the server sent an ECN-capable SYN/ACK, which was marked CE in the network, and the non-AccECN client fed this back by setting ECE on the ACK of the SYN/ACK.

5.2. Compatibility with TCP Experiments and Common TCP Options

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 3.2.3.3 provides guidance on how important it is to send an AccECN Option and whether it needs to be a full-length option.

Implementers of TFO need to take careful note of the recommendation in Section 3.2.2.1. That section recommends that, if the client has successfully negotiated AccECN, when acknowledging the SYN/ACK, even if it has data to send, it sends a pure ACK immediately before the data. Then it can reflect the IP-ECN field of the SYN/ACK on this pure ACK, which allows the server to detect ECN mangling.

5.3. Compatibility with Feedback Integrity Mechanisms

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 (similar to para 2 of Section 20.2 of [RFC3168]). Unlike the ECN Nonce [RFC3540], this approach does not waste the ECT(1) codepoint in the IP header, it does not require standardization 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 sparingly.

- Networks generate congestion signals when they are becoming congested, so networks 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 [RFC7713]. Whether the receiver or a downstream network is suppressing congestion feedback or the sender is unresponsive to the feedback, or both, ConEx audit can neutralize 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.

- 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.

Originally the ECN Nonce [RFC3540] was proposed to ensure integrity of congestion feedback. With minor changes AccECN could be optimized for the possibility that the ECT(1) codepoint might be used as an ECN Nonce. However, given RFC 3540 has been reclassified as historic, the AccECN design has been generalized 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.

6. 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. Within the constraints of the change-triggered ACK rules, the receiver can control how frequently it sends the AccECN TCP Option and therefore to some extent 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. And if data or ACKs are reordered, stale congestion information can be identified and ignored.

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 options are needed 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. Following ACK reordering, the Data Sender can reconstruct the order in which feedback was sent, but not until all the missing feedback has arrived.

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 3WHS 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. Then, the designers of security devices can understand which currently unused values might appear in future. So, even if they choose to treat such values as anomalous while they are not widely used, any blocking will at least be under policy control not hard-coded. Then, if previously unused values start to appear on the Internet (or in standards), such policies could be quickly reversed.

7. IANA Considerations

This document reassigns bit 7 of the TCP header flags to the AccECN experiment. This bit was previously called the Nonce Sum (NS) flag [RFC3540], but RFC 3540 has been reclassified as historic [RFC8311]. The flag will now be defined as:

Bit	Name	Reference
7	AE (Accurate ECN)	RFC XXXX

[TO BE REMOVED: IANA is requested to update the existing entry in the Transmission Control Protocol (TCP) Header Flags registration (https://www.iana.org/assignments/tcp-header-flags/tcp-header-flags.xhtml#tcp-header-flags-1) for Bit 7 to "AE (Accurate ECN), previously used as NS (Nonce Sum) by [RFC3540], which is now Historic [RFC8311]" and change the reference to this RFC-to-be instead of RFC8311.]

This document also defines two new TCP options for AccECN, assigned values of TBD0 and TBD1 (decimal) from the TCP option space. These values are defined as:

Kind	Length	Meaning	Reference
TBD0	N	Accurate ECN Order 0 (AccECN0)	RFC XXXX
TBD1	N	Accurate ECN Order 1 (AccECN1)	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 implementations using experimental option 254 per [RFC6994] with the single magic number 0xACCE (16 bits), as allocated in the IANA "TCP Experimental Option Experiment Identifiers (TCP ExIDs)" registry, SHOULD migrate to use these new option kinds (TBD0 & TBD1).

[TO BE REMOVED: The description of the 0xACCE value in the TCP ExIDs registry should be changed to "AccECN (current and new implementations SHOULD use option kinds TBD0 and TBD1)" at the following location: https://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml#tcp-exids]

8. 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 3.2.2.5). 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 3.2.2.5 and Appendix A.2).

Section 5.1 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. AccECN is compatible with the three schemes known to assure the integrity of ECN feedback (see Section 5.3 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.

There is a potential 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.

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.

9. Acknowledgements

We want to thank Koen De Schepper, Praveen Balasubramanian, Michael Welzl, Gorry Fairhurst, David Black, Spencer Dawkins, Michael Scharf, Michael Tuexen, Yuchung Cheng, Kenjiro Cho, Olivier Tilmans, Ilpo Jaervinen and Neal Cardwell 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 Comcast Innovation Fund, 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), and the Research Council of Norway through the TimeIn project. The views expressed here are solely those of the authors.

Mirja Kuehlewind was partly supported by the European Commission under Horizon 2020 grant agreement no. 688421 Measurement and Architecture for a Middleboxed Internet (MAMI), and by the Swiss State Secretariat for Education, Research, and Innovation under contract no. 15.0268. This support does not imply endorsement.

10. 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.

11. References

11.1. Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://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, <https://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, <https://www.rfc-editor.org/info/rfc5681>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

11.2. Informative References

[I-D.ietf-tcpm-2140bis]
Touch, J., Welzl, M., and S. Islam, "TCP Control Block Interdependence", draft-ietf-tcpm-2140bis-05 (work in progress), April 2020.

[I-D.ietf-tcpm-generalized-ecn]
Bagnulo, M. and B. Briscoe, "ECN++: Adding Explicit Congestion Notification (ECN) to TCP Control Packets", draft-ietf-tcpm-generalized-ecn-06 (work in progress), October 2020.

[I-D.ietf-tsvwg-l4s-arch]
Briscoe, B., Schepper, K., Bagnulo, M., and G. White, "Low Latency, Low Loss, Scalable Throughput (L4S) Internet Service: Architecture", draft-ietf-tsvwg-l4s-arch-07 (work in progress), October 2020.

[Mandalari18]
Mandalari, A., Lutu, A., Briscoe, B., Bagnulo, M., and Oe. Alay, "Measuring ECN++: Good News for ++, Bad News for ECN over Mobile", IEEE Communications Magazine , March 2018.

[RFC2018] Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, "TCP Selective Acknowledgment Options", RFC 2018, DOI 10.17487/RFC2018, October 1996, <https://www.rfc-editor.org/info/rfc2018>.

[RFC3449] Balakrishnan, H., Padmanabhan, V., Fairhurst, G., and M. Sooriyabandara, "TCP Performance Implications of Network Path Asymmetry", BCP 69, RFC 3449, DOI 10.17487/RFC3449, December 2002, <https://www.rfc-editor.org/info/rfc3449>.

[RFC3540] Spring, N., Wetherall, D., and D. Ely, "Robust Explicit Congestion Notification (ECN) Signaling with Nonces", RFC 3540, DOI 10.17487/RFC3540, June 2003, <https://www.rfc-editor.org/info/rfc3540>.

[RFC4987] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", RFC 4987, DOI 10.17487/RFC4987, August 2007, <https://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, <https://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, <https://www.rfc-editor.org/info/rfc5925>.

[RFC5961] Ramaiah, A., Stewart, R., and M. Dalal, "Improving TCP's Robustness to Blind In-Window Attacks", RFC 5961, DOI 10.17487/RFC5961, August 2010, <https://www.rfc-editor.org/info/rfc5961>.

[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, <https://www.rfc-editor.org/info/rfc6824>.

[RFC6994] Touch, J., "Shared Use of Experimental TCP Options", RFC 6994, DOI 10.17487/RFC6994, August 2013, <https://www.rfc-editor.org/info/rfc6994>.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <https://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, <https://www.rfc-editor.org/info/rfc7560>.

[RFC7713] Mathis, M. and B. Briscoe, "Congestion Exposure (ConEx) Concepts, Abstract Mechanism, and Requirements", RFC 7713, DOI 10.17487/RFC7713, December 2015, <https://www.rfc-editor.org/info/rfc7713>.

[RFC8257] Bensley, S., Thaler, D., Balasubramanian, P., Eggert, L., and G. Judd, "Data Center TCP (DCTCP): TCP Congestion Control for Data Centers", RFC 8257, DOI 10.17487/RFC8257, October 2017, <https://www.rfc-editor.org/info/rfc8257>.

[RFC8311] Black, D., "Relaxing Restrictions on Explicit Congestion Notification (ECN) Experimentation", RFC 8311, DOI 10.17487/RFC8311, January 2018, <https://www.rfc-editor.org/info/rfc8311>.

[RFC8511] Khademi, N., Welzl, M., Armitage, G., and G. Fairhurst, "TCP Alternative Backoff with ECN (ABE)", RFC 8511, DOI 10.17487/RFC8511, December 2018, <https://www.rfc-editor.org/info/rfc8511>.

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:

```
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

```
ECEB = r.ceb % DIVOPT
```

where '%' is the remainder operator.

On the arrival of an AccECN Option, the Data Sender first makes sure the ACK has not been superseded in order to avoid winding the s.ceb counter backwards. It uses the TCP acknowledgement number and any SACK options to calculate newlyAckedB, the amount of new data that the ACK acknowledges in bytes (newlyAckedB can be zero but not negative). If newlyAckedB is zero, either the ACK has been superseded or CE-marked packet(s) without data could have arrived. To break the tie for the latter case, the Data Sender could use timestamps (if present) to work out newlyAckedT, the amount of new time that the ACK acknowledges. If the Data Sender determines that the ACK has been superseded it ignores the AccECN Option. Otherwise, the Data Sender calculates the minimum non-negative difference d.ceb between the ECEB field and its local s.ceb counter, using modulo arithmetic as follows:

```
if ((newlyAckedB > 0) || (newlyAckedT > 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 3.2.2.5 and Section 3.2.3.2); 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:

```
DIVACE = 2^3
```

Every time an Acceptable CE marked packet arrives (Section 3.2.2.2), 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

```
ACE = r.cep % 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 ensures the ACK has not been superseded using the TCP acknowledgement number, any SACK options and timestamps (if available) to calculate newlyAckedB, as in Appendix A.1. If the ACK has not been superseded, 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 ((newlyAckedB > 0) || (newlyAckedT > 0))
    d.cep = (ACE + DIVACE - (s.cep % DIVACE)) % DIVACE
```

Section 3.2.2.5 expects the Data Sender to assume that the ACE field cycled if it is the safest likely case 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 missing ACKs were piggy-backed on data (i.e. not pure ACKs) retransmissions will not repair the lost AccECN information, because AccECN requires retransmissions to carry the latest AccECN counters, not the original ones.

The phrase 'under prevailing conditions' allows for implementation-dependent interpretation. A Data Sender might take account of the prevailing size of data segments and the prevailing CE marking rate just before the sequence of missing ACKs. 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 newlyAckedB is the amount of data that an ACK acknowledges since the previous ACK, then the Data Sender could assume that this acknowledges newlyAckedPkt full-sized segments, where newlyAckedPkt = newlyAckedB/MSS. Then it could assume that the ACE field incremented by

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

For example, imagine an ACK acknowledges newlyAckedPkt=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).

ACKs that acknowledge a large stretch of packets might be common in data centres to achieve a high packet rate or might be due to ACK thinning by a middlebox. In these cases, cycling of the ACE field would often appear to have been possible, so the above algorithm would be over-conservative, leading to a false high marking rate and poor performance. Therefore it would be reasonable to only use dSafer.cep rather than d.cep if the moving average of newlyAckedPkt was well below 8.

Implementers could build in more heuristics to estimate prevailing average segment size and prevailing ECN marking. For instance, newlyAckedPkt in the above formula could be replaced with newlyAckedPktHeur = newlyAckedPkt*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 3.2.2.5 says "the Data Sender MAY attempt to neutralize 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 AccECN Option

When the AccECN 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 AccECN Option without needing to process the ACE field. If for some reason it needs CE-marked packets, if dSafer.cep is different from d.cep, it can determine whether d.cep is likely to be a safe enough estimate by checking whether the average marked segment size (s = d.ceb/d.cep) is less than the MSS (where d.ceb is the amount of newly CE-marked bytes - see Appendix A.1). Specifically, it could use the following algorithm:

```
      SAFETY_FACTOR = 2
      if (dSafer.cep > d.cep) {
          if (d.ceb <= MSS * d.cep) {  % Same as (s <= MSS), but no DBZ
             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:

```
                    ^
              sSafer|
                    |
                 MSS+
                    |
                    |         dSafer.cep
                    |                  is
   MSS/SAFETY_FACTOR+--------------+    safest
                    |              |
                    | d.cep is safe|
                    |    enough    |
                    +-------------------->
                                  MSS     s
```

The following examples give the reasoning behind the algorithm, assuming MSS=1460 [B]:

- o if d.cep=0, dSafer.cep=8 and d.ceb=1460, 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=1460, 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=10200, then s=1457 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. Some possible ways to calculate s_ave are outlined below. The precise details will depend on why an estimate of marked bytes is needed.

The implementation 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 is reset once per RTT. Either way, it would estimate s_ave as:

```
s_ave ~= flightsize / 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 3.2.3.3 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 optimized 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()
```

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. In this case few options will be sent (evtl. even less than 3 per RTT). However, when continuously sending data, data packets as well as ACKs will spread out equally over the RTT and sufficient ACKs with the AccECN option will be sent.

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.e1b. Note that, because r.e0b is initialized to 1 and the other two counters are initialized 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 retransmissions 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. Rationale for Usage of TCP Header Flags

B.1. Three TCP Header Flags in the SYN-SYN/ACK Handshake

AccECN uses a rather unorthodox approach to negotiate the highest version TCP ECN feedback scheme that both ends support, as justified below. It follows from the original TCP ECN capability negotiation [RFC3168], in which the client set the 2 least significant of the original reserved flags in the TCP header, and fell back to no ECN support if the server responded with the 2 flags cleared, which had previously been the default.

ECN originally used header flags rather than a TCP option because it was considered more efficient to use a header flag for 1 bit of feedback per ACK, and this bit could be overloaded to indicate support for ECN during the handshake. During the development of ECN, 1 bit crept up to 2, in order to deliver the feedback reliably and to work round some broken hosts that reflected the reserved flags during the handshake.

In order to be backward compatible with RFC 3168, AccECN continues this approach, using the 3rd least significant TCP header flag that had previously been allocated for the ECN nonce (now historic). Then, whatever form of server an AccECN client encounters, the connection can fall back to the highest version of feedback protocol that both ends support, as explained in Section 3.1.

If AccECN had used the more orthodox approach of a TCP option, it would still have had to set the two ECN flags in the main TCP header, in order to be able to fall back to Classic RFC 3168 ECN, or to disable ECN support, without another round of negotiation. Then AccECN would also have had to handle all the different ways that servers currently respond to settings of the ECN flags in the main TCP header, including all the conflicting cases where a server might have said it supported one approach in the flags and another approach in the new TCP option. And AccECN would have had to deal with all the additional possibilities where a middlebox might have mangled the ECN flags, or removed the TCP option. Thus, usage of the 3rd reserved TCP header flag simplified the protocol.

The third flag was used in a way that could be distinguished from the ECN nonce, in case any nonce deployment was encountered. Previous usage of this flag for the ECN nonce was integrated into the original ECN negotiation. This further justified the 3rd flag's use for

AccECN, because a non-ECN usage of this flag would have had to use it as a separate single bit, rather than in combination with the other 2 ECN flags.

Indeed, having overloaded the original uses of these three flags for its handshake, AccECN overloads all three bits again as a 3-bit counter.

B.2. Four Codepoints in the SYN/ACK

Of the 8 possible codepoints that the 3 TCP header flags can indicate on the SYN/ACK, 4 already indicated earlier (or broken) versions of ECN support. In the early design of AccECN, an AccECN server could use only 2 of the 4 remaining codepoints. They both indicated AccECN support, but one fed back that the SYN had arrived marked as CE. Even though ECN support on a SYN is not yet on the standards track, the idea is for either end to act as a dumb reflector, so that future capabilities can be unilaterally deployed without requiring 2-ended deployment (justified in Section 2.5).

During traversal testing it was discovered that the ECN field in the SYN was mangled on a non-negligible proportion of paths. Therefore it was necessary to allow the SYN/ACK to feed all four IP/ECN codepoints that the SYN could arrive with back to the client. Without this, the client could not know whether to disable ECN for the connection due to mangling of the IP/ECN field (also explained in Section 2.5). This development consumed the remaining 2 codepoints on the SYN/ACK that had been reserved for future use by AccECN in earlier versions.

B.3. Space for Future Evolution

Despite availability of usable TCP header space being extremely scarce, the AccECN protocol has taken all possible steps to ensure that there is space to negotiate possible future variants of the protocol, either if the experiment proves that a variant of AccECN is required, or if a completely different ECN feedback approach is needed:

Future AccECN variants: When the AccECN capability is negotiated during TCP's 3WHS, the rows in Table 2 tagged as 'Nonce' and 'Broken' in the column for the capability of node B are unused by any current protocol in the RFC series. These could be used by TCP servers in future to indicate a variant of the AccECN protocol. In recent measurement studies in which the response of large numbers of servers to an AccECN SYN has been tested, e.g. [Mandalari18], a very small number of SYN/ACKs arrive with the pattern tagged as 'Nonce', and a small but more significant number arrive with the pattern tagged as 'Broken'. The 'Nonce' pattern could be a sign that a few servers have implemented the ECN Nonce [RFC3540], which has now been reclassified as historic [RFC8311], or it could be the random result of some unknown middlebox behaviour. The greater prevalence of the 'Broken' pattern suggests that some instances still exist of the broken code that reflects the reserved flags on the SYN.

The requirement not to reject unexpected initial values of the ACE counter (in the main TCP header) in the last para of Section 3.2.2.3 ensures that 3 unused codepoints on the ACK of the SYN/ACK, 6 unused values on the first SYN=0 data packet from the client and 7 unused values on the first SYN=0 data packet from the server could be used to declare future variants of the AccECN protocol. The word 'declare' is used rather than 'negotiate' because, at this late stage in the 3WHS, it would be too late for a negotiation between the endpoints to be completed. A similar requirement not to reject unexpected initial values in the TCP option (Section 3.2.3.2.4) is for the same purpose. If traversal of the TCP option were reliable, this would have enabled a far wider range of future variation of the whole AccECN protocol. Nonetheless, it could be used to reliably negotiate a wide range of variation in the semantics of the AccECN Option.

Future non-AccECN variants: Five codepoints out of the 8 possible in the 3 TCP header flags used by AccECN are unused on the initial SYN (in the order AE,CWR,ECE): 001, 010, 100, 101, 110. Section 3.1.3 ensures that the installed base of AccECN servers will all assume these are equivalent to AccECN negotiation with 111 on the SYN. These codepoints would not allow fall-back to Classic ECN support for a server that did not understand them, but this approach ensures they are available in future, perhaps for uses other than ECN alongside the AccECN scheme. All possible combinations of SYN/ACK could be used in response except either 000 or reflection of the same values sent on the SYN.

Of course, other ways could be resorted to in order to extend AccECN or ECN in future, although their traversal properties are likely to be inferior. They include a new TCP option; using the remaining reserved flags in the main TCP header (preferably extending the 3-bit combinations used by AccECN to 4-bit combinations, rather than burning one bit for just one state); a non-zero urgent pointer in combination with the URG flag cleared; or some other unexpected combination of fields yet to be invented.

Authors' Addresses

Bob Briscoe
Independent
UK

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

Mirja Kuehlewind
Ericsson
Germany

EMail: ietf@kuehlewind.net

Richard Scheffenegger
NetApp
Vienna
Austria

EMail: Richard.Scheffenegger@netapp.com

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: August 18, 2018

N. Khademi
M. Welzl
University of Oslo
G. Armitage
Swinburne University of Technology
G. Fairhurst
University of Aberdeen
February 14, 2018

TCP Alternative Backoff with ECN (ABE)
draft-ietf-tcpm-alternativebackoff-ecn-06

Abstract

Active Queue Management (AQM) mechanisms allow for burst tolerance while enforcing short queues to minimise the time that packets spend enqueued at a bottleneck. This can cause noticeable performance degradation for TCP connections traversing such a bottleneck, especially if there are only a few flows or their bandwidth-delay-product is large. An Explicit Congestion Notification (ECN) signal indicates that an AQM mechanism is used at the bottleneck, and therefore the bottleneck network queue is likely to be short. This document therefore proposes an update to RFC3168, which changes the TCP sender-side ECN reaction in congestion avoidance to reduce the Congestion Window (cwnd) by a smaller amount than the congestion control algorithm's reaction to inferred packet loss.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Explicit Congestion Notification (ECN) [RFC3168] makes it possible for an Active Queue Management (AQM) mechanism to signal the presence of incipient congestion without incurring packet loss. This lets the network deliver some packets to an application that would have been dropped if the application or transport did not support ECN. This packet loss reduction is the most obvious benefit of ECN, but it is often relatively modest. Other benefits of deploying ECN have been documented in RFC8087 [RFC8087].

The rules for ECN were originally written to be very conservative, and required the congestion control algorithms of ECN-Capable transport protocols to treat ECN congestion signals exactly the same as they would treat an inferred packet loss [RFC3168].

Research has demonstrated the benefits of reducing network delays that are caused by interaction of loss-based TCP congestion control and excessive buffering [BUFFERBLOAT]. This has led to the creation of new AQM mechanisms like PIE [RFC8033] and CoDel [CODEL2012][I-D.CoDel], which prevent bloated queues that are common with unmanaged and excessively large buffers deployed across the Internet [BUFFERBLOAT].

The AQM mechanisms mentioned above aim to keep a sustained queue short while tolerating transient (short-term) packet bursts. However, currently used loss-based congestion control mechanisms cannot always utilise a bottleneck link well where there are short queues. For example, a TCP sender must be able to store at least an end-to-end bandwidth-delay product (BDP) worth of data at the bottleneck buffer if it is to maintain full path utilisation in the face of loss-induced reduction of cwnd [RFC5681], which effectively doubles the amount of data that can be in flight, the maximum round-trip time (RTT) experience, and the path’s effective RTT using the network path.

Modern AQM mechanisms can use ECN to signal the early signs of impending queue buildup long before a tail-drop queue would be forced to resort to dropping packets. It is therefore appropriate for the transport protocol congestion control algorithm to have a more measured response when an early-warning signal of congestion is received in the form of an ECN CE-marked packet. Recognizing these changes in modern AQM practices, more recent rules have relaxed the strict requirement that ECN signals be treated identically to inferred packet loss [I-D.ECN-exp]. Following these newer, more flexible rules, this document defines a new sender-side-only congestion control response, called "ABE" (Alternative Backoff with ECN). ABE improves TCP’s average throughput when routers use AQM controlled buffers that allow for short queues only.

2. Definitions

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].

3. Specification

This specification updates the congestion control algorithm of an ECN-Capable TCP transport protocol by changing the TCP sender response to feedback from the TCP receiver that indicates reception of a CE-marked packet, i.e., receipt of a packet with the ECN-Echo flag (defined in [RFC3168]) set.

It updates the following text in section 6.1.2 of the ECN specification [RFC3168] :

> The indication of congestion should be treated just as a congestion loss in non-ECN-Capable TCP. That is, the TCP source halves the congestion window "cwnd" and reduces the slow start threshold "ssthresh".

Replacing this with:

> Receipt of a packet with the ECN-Echo flag SHOULD trigger the TCP source to set the slow start threshold (ssthresh) to 0.8 times the FlightSize, with a lower bound of 2 * SMSS applied to the result. As in [RFC5681], the TCP sender also reduces the cwnd value to no more than the new ssthresh value.

4. Discussion

Much of the technical background to ABE can be found in a research paper [ABE2017]. This paper used a mix of experiments, theory and simulations with NewReno [RFC5681] and CUBIC [I-D.CUBIC] to evaluate the technique. The technique was shown to present "...significant performance gains in lightly-multiplexed [few concurrent flows] scenarios, without losing the delay-reduction benefits of deploying CoDel or PIE". The performance improvement is achieved when reacting to ECN-Echo in congestion avoidance by multiplying cwnd and ssthresh with a value in the range [0.7,0.85].

4.1. Why Use ECN to Vary the Degree of Backoff?

The classic rule-of-thumb dictates that a network path needs to provide a BDP of bottleneck buffering if a TCP connection wishes to optimise path utilisation. A single TCP bulk transfer running through such a bottleneck will have increased its congestion window (cwnd) up to 2*BDP by the time that packet loss occurs. According to [RFC5681], when a TCP sender detects segment loss using the retransmission timer and the given segment has not yet been resent by way of the retransmission timer, the value of ssthresh must be set to no more of the maximum of half of the FlightSize and 2*SMSS. The same equation is also used during Fast Retransmit/Fast Recovery [RFC5681]. As a result, the TCP congestion control only allows one BDP of packets in flight. This is just sufficient to maintain 100% utilisation of the bottleneck on the network path.

AQM mechanisms such as CoDel [I-D.CoDel] and PIE [RFC8033] set a delay target in routers and use congestion notifications to constrain the queuing delays experienced by packets, rather than in response to impending or actual bottleneck buffer exhaustion. With current default delay targets, CoDel and PIE both effectively emulate a bottleneck with a short queue (section II, [ABE2017]) while also allowing short traffic bursts into the queue. This provides acceptable performance for TCP connections over a path with a low BDP, or in highly multiplexed scenarios (many concurrent transport flows). However, in a lightly-multiplexed case over a path with a large BDP, conventional TCP backoff leads to gaps in packet transmission and under-utilisation of the path.

Instead of discarding packets, an AQM mechanism is allowed to mark ECN-Capable packets with an ECN CE-mark. The reception of a CE-mark feedback not only indicates congestion on the network path, it also indicates that an AQM mechanism exists at the bottleneck along the path, and hence the CE-mark likely came from a bottleneck with a controlled short queue. Reacting differently to an ECN-signalled congestion than to an inferred packet loss can then yield the benefit of a reduced back-off when queues are short. Using ECN can also be advantageous for several other reasons [RFC8087].

The idea of reacting differently to inferred packet loss and detection of an ECN-signalled congestion pre-dates this document. For example, previous research proposed using ECN CE-marked feedback to modify TCP congestion control behaviour via a larger multiplicative decrease factor in conjunction with a smaller additive increase factor [ICC2002]. The goal of this former work was to operate across AQM bottlenecks using Random Early Detection (RED) that were not necessarily configured to emulate a short queue (The current usage of RED as an Internet AQM method is limited [RFC7567]).

4.2. Focus on ECN as Defined in RFC3168

Some transport protocol mechanisms rely on ECN semantics that differ from the original ECN definition [RFC3168]. For instance, Accurate ECN [I-D.ietf-tcpm-accurate-ecn] permits more frequent and detailed feedback. Use of mechanisms (such as Accurate ECN, Datacenter TCP (DCTCP) [RFC8257], or Congestion Exposure (ConEx) [RFC7713]) is out of scope for this document. This specification focuses on ECN as defined in [RFC3168].

4.3. Choice of ABE Multiplier

ABE decouples the reaction of a TCP sender to inferred packet loss and ECN-signalled congestion in the congestion avoidance phase. To achieve this, ABE uses different the scaling factor in Equation 4 in

Section 3.1 of [RFC5681]. The description respectively uses beta_{loss} and beta_{ecn} to refer to the multiplicative decrease factors applied in response to inferred packet loss, and in response to a receiver indicating ECN-signalled congestion. For non-ECN-enabled TCP connections, only beta_{loss} applies.

In other words, in response to inferred packet loss:

ssthresh = max (FlightSize * beta_{loss}, 2 * SMSS)

and in response to an indication of an ECN-signalled congestion:

ssthresh = max (FlightSize * beta_{ecn}, 2 * SMSS)

and

cwnd = ssthresh

where FlightSize is the amount of outstanding data in the network, upper-bounded by the smaller of the sender’s cwnd and the receiver’s advertised window (rwnd) [RFC5681]. The higher the values of beta_{loss} and beta_{ecn}, the less aggressive the response of any individual backoff event.

The appropriate choice for beta_{loss} and beta_{ecn} values is a balancing act between path utilisation and draining the bottleneck queue. More aggressive backoff (smaller beta_*) risks underutilising the path, while less aggressive backoff (larger beta_*) can result in slower draining of the bottleneck queue.

The Internet has already been running with at least two different beta_{loss} values for several years: the standard value is 0.5 [RFC5681], and the Linux implementation of CUBIC [I-D.CUBIC] has used a multiplier of 0.7 since kernel version 2.6.25 released in 2008. ABE proposes no change to beta_{loss} used by current TCP implementations.

The recommendation in Section 3 in this document corresponds to a value of beta_{ecn}=0.8. This recommended beta_{ecn} value is only applicable for the standard TCP congestion control [RFC5681]. The selection of beta_{ecn} enables tuning the response of a TCP connection to shallow AQM marking thresholds. beta_{loss} characterizes the response of a congestion control algorithm to packet loss, i.e., exhaustion of buffers (of unknown depth). Different values for beta_{loss} have been suggested for TCP congestion control algorithms. Consequently, beta_{ecn} is likely to be an algorithm-specific parameter rather than a constant multiple of the algorithm’s existing beta_{loss}.

A range of tests (section IV, [ABE2017]) with NewReno and CUBIC over CoDel and PIE in lightly-multiplexed scenarios have explored this choice of parameter. The results of these tests indicate that CUBIC connections benefit from beta_{ecn} of 0.85 (cf. beta_{loss} = 0.7), and NewReno connections see improvements with beta_{ecn} in the range 0.7 to 0.85 (cf. beta_{loss} = 0.5).

5. ABE Deployment Requirements

This update is a sender-side only change. Like other changes to congestion control algorithms, it does not require any change to the TCP receiver or to network devices. It does not require any ABE-specific changes in routers or the use of Accurate ECN feedback [I-D.ietf-tcpm-accurate-ecn] by a receiver.

RFC3168 states that the congestion control response to an ECN-signalled congestion is the same as the response to a dropped packet [RFC3168]. [I-D.ECN-exp] updates this specification to allow systems to provide a different behaviour when they experience ECN-signalled congestion rather than packet loss. The present specification defines such an experiment and has thus been assigned an Experimental status before being proposed as a Standards-Track update.

The purpose of the Internet experiment is to collect experience with deployment of ABE, and confirm the safety in deployed networks using this update to TCP congestion control.

When used with bottlenecks that do not support ECN-marking the specification does not modify the transport protocol.

To evaluate the benefit, this experiment therefore requires support in AQM routers for ECN-marking of packets carrying the ECN-Capable Transport, ECT(0), codepoint [RFC3168].

If the method is only deployed by some senders, and not by others, the senders that use this method can gain some advantage, possibly at the expense of other flows that do not use this updated method. Because this advantage applies only to ECN-marked packets and not to packet loss indications, an ECN-Capable bottleneck will still fall back to dropping packets if an TCP sender using ABE is too aggressive, and the result is no different than if the TCP sender was using traditional loss-based congestion control.

A TCP sender reacts to loss or ECN marks only once per round-trip time. Hence, if a sender would first be notified of an ECN mark and then learn about loss in the same round-trip, it would only react to the first notification (ECN) but not to the second (loss). RFC3168 specified a reaction to ECN that was equal to the reaction to loss [RFC3168].

ABE also responds to congestion once per RTT, and therefore it does not respond to further loss within the same RTT, since ABE has already reduced the congestion window. If congestion persists after such reduction, ABE continues to reduce the congestion window in each consecutive RTT. This consecutive reduction can protect the network against long-standing unfairness in the case of AQM algorithms that do not keep a small average queue length.

The result of this Internet experiment ought to include an investigation of the implications of experiencing an ECN-CE mark followed by loss within the same RTT. At the end of the experiment, this will be reported to the TCPM WG (or IESG).

6. Acknowledgements

Authors N. Khademi, M. Welzl and G. Fairhurst were part-funded by the European Community under its Seventh Framework Programme through the Reducing Internet Transport Latency (RITE) project (ICT-317700). The views expressed are solely those of the authors.

The authors would like to thank Stuart Cheshire for many suggestions when revising the draft, and the following people for their contributions to [ABE2017]: Chamil Kulatunga, David Ros, Stein Gjessing, Sebastian Zander. Thanks also to (in alphabetical order) Roland Bless, Bob Briscoe, David Black, Markku Kojo, John Leslie, Lawrence Stewart, Dave Taht and the TCPM working group for providing valuable feedback on this document.

The authors would finally like to thank everyone who provided feedback on the congestion control behaviour specified in this update received from the IRTF Internet Congestion Control Research Group (ICCRG).

7. IANA Considerations

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

This document includes no request to IANA.

8. Implementation Status

ABE is implemented as a patch for Linux and FreeBSD. It is meant for research and available for download from http://heim.ifi.uio.no/naeemk/research/ABE/. This code was used to produce the test results that are reported in [ABE2017]. An evolved version of the patch for FreeBSD is currently under review for potential inclusion in the mainline kernel [ABE-FreeBSD].

9. Security Considerations

The described method is a sender-side only transport change, and does not change the protocol messages exchanged. The security considerations for ECN [RFC3168] therefore still apply.

This is a change to TCP congestion control with ECN that will typically lead to a change in the capacity achieved when flows share a network bottleneck. This could result in some flows receiving more than their fair share of capacity. Similar unfairness in the way that capacity is shared is also exhibited by other congestion control mechanisms that have been in use in the Internet for many years (e.g., CUBIC [I-D.CUBIC]). Unfairness may also be a result of other factors, including the round trip time experienced by a flow. ABE applies only when ECN-marked packets are received, not when packets are lost, hence use of ABE cannot lead to congestion collapse.

10. Revision Information

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

-06. Addressed Michael Scharf's comments.

-05. Refined the description of the experiment based on feedback at IETF-100. Incorporated comments from David Black.

-04. Incorporates review comments from Lawrence Stewart and the remaining comments from Roland Bless. References are updated.

-03. Several review comments from Roland Bless are addressed. Consistent terminology and equations. Clarification on the scope of recommended beta_{ecn} value.

-02. Corrected the equations in Section 4.3. Updated the affiliations. Lower bound for cwnd is defined. A recommendation for window-based transport protocols is changed to cover all transport protocols that implement a congestion control reduction to an ECN congestion signal. Added text about ABE's FreeBSD mainline kernel status including a reference to the FreeBSD code review page. References are updated.

-01. Text improved, mainly incorporating comments from Stuart Cheshire. The reference to a technical report has been updated to a published version of the tests [ABE2017]. Used "AQM Mechanism" throughout in place of other alternatives, and more consistent use of technical language and clarification on the intended purpose of the experiments required by EXP status. There was no change to the technical content.

-00. draft-ietf-tcpm-alternativebackoff-ecn-00 replaces draft-khademi-tcpm-alternativebackoff-ecn-01. Text describing the nature of the experiment was added.

Individual draft -01. This I-D now refers to draft-black-tsvwg-ecn-experimentation-02, which replaces draft-khademi-tsvwg-ecn-response-00 to make a broader update to RFC3168 for the sake of allowing experiments. As a result, some of the motivating and discussing text that was moved from draft-khademi-alternativebackoff-ecn-03 to draft-khademi-tsvwg-ecn-response-00 has now been re-inserted here.

Individual draft -00. draft-khademi-tsvwg-ecn-response-00 and draft-khademi-tcpm-alternativebackoff-ecn-00 replace draft-khademi-alternativebackoff-ecn-03, following discussion in the TSVWG and TCPM working groups.

11. References

11.1. Normative References

[I-D.ECN-exp] Black, D., "Explicit Congestion Notification (ECN) Experimentation", Internet-draft, IETF work-in-progress draft-ietf-tsvwg-ecn-experimentation-08, November 2017.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://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, <https://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, <https://www.rfc-editor.org/info/rfc5681>.

[RFC7567] Baker, F., Ed. and G. Fairhurst, Ed., "IETF Recommendations Regarding Active Queue Management", BCP 197, RFC 7567, DOI 10.17487/RFC7567, July 2015, <https://www.rfc-editor.org/info/rfc7567>.

[RFC8257] Bensley, S., Thaler, D., Balasubramanian, P., Eggert, L., and G. Judd, "Data Center TCP (DCTCP): TCP Congestion Control for Data Centers", RFC 8257, DOI 10.17487/RFC8257, October 2017, <https://www.rfc-editor.org/info/rfc8257>.

11.2. Informative References

[ABE-FreeBSD] "ABE patch review in FreeBSD", <https://reviews.freebsd.org/D11616>.

[ABE2017] Khademi, N., Armitage, G., Welzl, M., Fairhurst, G., Zander, S., and D. Ros, "Alternative Backoff: Achieving Low Latency and High Throughput with ECN and AQM", IFIP NETWORKING 2017, Stockholm, Sweden, June 2017.

[BUFFERBLOAT] Gettys, J. and K. Nichols, "Bufferbloat: Dark Buffers in the Internet", November 2011.

[CODEL2012] Nichols, K. and V. Jacobson, "Controlling Queue Delay", July 2012, <http://queue.acm.org/detail.cfm?id=2209336>.

[I-D.CoDel] Nichols, K., Jacobson, V., McGregor, V., and J. Iyengar, "Controlled Delay Active Queue Management", Internet-draft, IETF work-in-progress draft-ietf-aqm-codel-10, October 2017.

[I-D.CUBIC] Rhee, I., Xu, L., Ha, S., Zimmermann, A., Eggert, L., and R. Scheffenegger, "CUBIC for Fast Long-Distance Networks", Internet-draft, IETF work-in-progress draft-ietf-tcpm-cubic-07, November 2017.

[I-D.ietf-tcpm-accurate-ecn] Briscoe, B., Kuehlewind, M., and R. Scheffenegger, "More Accurate ECN Feedback in TCP", draft-ietf-tcpm-accurate-ecn-03 (work in progress), May 2017.

[ICC2002] Kwon, M. and S. Fahmy, "TCP Increase/Decrease Behavior with Explicit Congestion Notification (ECN)", IEEE ICC 2002, New York, New York, USA, May 2002, <http://dx.doi.org/10.1109/ICC.2002.997262>.

[RFC7713] Mathis, M. and B. Briscoe, "Congestion Exposure (ConEx) Concepts, Abstract Mechanism, and Requirements", RFC 7713, DOI 10.17487/RFC7713, December 2015, <https://www.rfc-editor.org/info/rfc7713>.

[RFC8033] Pan, R., Natarajan, P., Baker, F., and G. White, "Proportional Integral Controller Enhanced (PIE): A Lightweight Control Scheme to Address the Bufferbloat Problem", RFC 8033, DOI 10.17487/RFC8033, February 2017, <https://www.rfc-editor.org/info/rfc8033>.

[RFC8087] Fairhurst, G. and M. Welzl, "The Benefits of Using Explicit Congestion Notification (ECN)", RFC 8087, DOI 10.17487/RFC8087, March 2017, <https://www.rfc-editor.org/info/rfc8087>.

Authors' Addresses

Naeem Khademi
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Email: naeemk@ifi.uio.no

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

Email: michawe@ifi.uio.no

Grenville Armitage
Internet For Things (I4T) Research Group
Swinburne University of Technology
PO Box 218
John Street, Hawthorn
Victoria 3122
Australia

Email: garmitage@swin.edu.au

Godred Fairhurst
University of Aberdeen
School of Engineering, Fraser Noble Building
Aberdeen AB24 3UE
UK

Email: gorry@erg.abdn.ac.uk

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: March 18, 2019

N. Khademi
M. Welzl
University of Oslo
G. Armitage
Netflix
G. Fairhurst
University of Aberdeen
September 14, 2018

TCP Alternative Backoff with ECN (ABE)
draft-ietf-tcpm-alternativebackoff-ecn-12

Abstract

Active Queue Management (AQM) mechanisms allow for burst tolerance while enforcing short queues to minimise the time that packets spend enqueued at a bottleneck. This can cause noticeable performance degradation for TCP connections traversing such a bottleneck, especially if there are only a few flows or their bandwidth-delay-product is large. The reception of a Congestion Experienced (CE) ECN mark indicates that an AQM mechanism is used at the bottleneck, and therefore the bottleneck network queue is likely to be short. Feedback of this signal allows the TCP sender-side ECN reaction in congestion avoidance to reduce the Congestion Window (cwnd) by a smaller amount than the congestion control algorithm’s reaction to inferred packet loss. This specification therefore defines an experimental change to the TCP reaction specified in RFC3168, as permitted by RFC 8311.

Copyright Notice

Table of Contents

1. Introduction

Explicit Congestion Notification (ECN) [RFC3168] makes it possible for an Active Queue Management (AQM) mechanism to signal the presence of incipient congestion without necessarily incurring packet loss. This lets the network deliver some packets to an application that would have been dropped if the application or transport did not support ECN. This packet loss reduction is the most obvious benefit of ECN, but it is often relatively modest. Other benefits of deploying ECN have been documented in RFC8087 [RFC8087].

The rules for ECN were originally written to be very conservative, and required the congestion control algorithms of ECN-Capable transport protocols to treat indications of congestion signalled by ECN exactly the same as they would treat an inferred packet loss [RFC3168]. Research has demonstrated the benefits of reducing network delays that are caused by interaction of loss-based TCP congestion control and excessive buffering [BUFFERBLOAT]. This has led to the creation of AQM mechanisms like Proportional Integral Controller Enhanced (PIE) [RFC8033] and Controlling Queue Delay (CoDel) [CODEL2012][RFC8289], which prevent bloated queues that are common with unmanaged and excessively large buffers deployed across the Internet [BUFFERBLOAT].

The AQM mechanisms mentioned above aim to keep a sustained queue short while tolerating transient (short-term) packet bursts. However, currently used loss-based congestion control mechanisms are not always able to effectively utilise a bottleneck link where there are short queues. For example, a TCP sender using the Reno congestion control needs to be able to store at least an end-to-end bandwidth-delay product (BDP) worth of data at the bottleneck buffer if it is to maintain full path utilisation in the face of loss-induced reduction of the congestion window (cwnd) [RFC5681]. This amount of buffering effectively doubles the amount of data that can be in flight and the maximum round-trip time (RTT) experienced by the TCP sender.

Modern AQM mechanisms can use ECN to signal the early signs of impending queue buildup long before a tail-drop queue would be forced to resort to dropping packets. It is therefore appropriate for the transport protocol congestion control algorithm to have a more measured response when it receives an indication with an early-warning of congestion after the remote endpoint receives an ECN CE-marked packet. Recognizing these changes in modern AQM practices, the strict requirement that ECN CE signals be treated identically to inferred packet loss has been relaxed [RFC8311]. This document therefore defines a new sender-side-only congestion control response, called "ABE" (Alternative Backoff with ECN). ABE improves TCP’s average throughput when routers use AQM controlled buffers that allow only for short queues.

2. Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 RFC 2119 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Specification

This specification changes the congestion control algorithm of an ECN-Capable TCP transport protocol by changing the TCP sender response to feedback from the TCP receiver that indicates reception of a CE-marked packet, i.e., receipt of a packet with the ECN-Echo flag (defined in [RFC3168]) set, following the process defined in [RFC8311].

The TCP sender response is currently specified in section 6.1.2 of the ECN specification [RFC3168], updated by [RFC8311]:

> The indication of congestion should be treated just as a congestion loss in non-ECN-Capable TCP. That is, the TCP source halves the congestion window "cwnd" and reduces the slow start threshold "ssthresh", unless otherwise specified by an Experimental RFC in the IETF document stream.

Following publication of RFC 8311, this document specifies a sender-side change to TCP:

> Receipt of a packet with the ECN-Echo flag SHOULD trigger the TCP source to set the slow start threshold (ssthresh) to 0.8 times the FlightSize, with a lower bound of 2 * SMSS applied to the result. As in [RFC5681], the TCP sender also reduces the cwnd value to no more than the new ssthresh value. RFC 3168 section 6.1.2 provides guidance on setting a cwnd less than 2 * SMSS.

3.1. Choice of ABE Multiplier

ABE decouples the reaction of a TCP sender to inferred packet loss and indication of ECN-signalled congestion in the congestion avoidance phase. To achieve this, ABE uses a different scaling factor in Equation 4 in Section 3.1 of [RFC5681]. The description respectively uses beta_{loss} and beta_{ecn} to refer to the multiplicative decrease factors applied in response to inferred packet loss, and in response to a receiver indicating ECN-signalled congestion. For non-ECN-enabled TCP connections, only beta_{loss} applies.

In other words, in response to inferred packet loss:

```
ssthresh = max (FlightSize * beta_{loss}, 2 * SMSS)
```

and in response to an indication of an ECN-signalled congestion:

```
ssthresh = max (FlightSize * beta_{ecn}, 2 * SMSS)
```

and

cwnd = ssthresh

(If ssthresh == 2 * SMSS, RFC 3168 section 6.1.2 provides guidance on setting a cwnd lower than 2 * SMSS.)

where FlightSize is the amount of outstanding data in the network, upper-bounded by the smaller of the sender’s cwnd and the receiver’s advertised window (rwnd) [RFC5681]. The higher the values of beta_{loss} and beta_{ecn}, the less aggressive the response of any individual backoff event.

The appropriate choice for beta_{loss} and beta_{ecn} values is a balancing act between path utilisation and draining the bottleneck queue. More aggressive backoff (smaller beta_*) risks underutilising the path, while less aggressive backoff (larger beta_*) can result in slower draining of the bottleneck queue.

The Internet has already been running with at least two different beta_{loss} values for several years: the standard value is 0.5 [RFC5681], and the Linux implementation of CUBIC [RFC8312] has used a multiplier of 0.7 since kernel version 2.6.25 released in 2008. ABE does not change the value of beta_{loss} used by current TCP implementations.

The recommendation in this document specifies a value of beta_{ecn}=0.8. This recommended beta_{ecn} value is only applicable for the standard TCP congestion control [RFC5681]. The selection of beta_{ecn} enables tuning the response of a TCP connection to shallow AQM marking thresholds. beta_{loss} characterizes the response of a congestion control algorithm to packet loss, i.e., exhaustion of buffers (of unknown depth). Different values for beta_{loss} have been suggested for TCP congestion control algorithms. Consequently, beta_{ecn} is likely to be an algorithm-specific parameter rather than a constant multiple of the algorithm’s existing beta_{loss}.

A range of tests (section IV, [ABE2017]) with NewReno and CUBIC over CoDel and PIE in lightly-multiplexed scenarios have explored this choice of parameter. The results of these tests indicate that CUBIC connections benefit from beta_{ecn} of 0.85 (cf. beta_{loss} = 0.7), and NewReno connections see improvements with beta_{ecn} in the range 0.7 to 0.85 (cf. beta_{loss} = 0.5).

4. Discussion

Much of the technical background to ABE can be found in a research paper [ABE2017]. This paper used a mix of experiments, theory and simulations with NewReno [RFC5681] and CUBIC [RFC8312] to evaluate the technique. The technique was shown to present "...significant performance gains in lightly-multiplexed [few concurrent flows] scenarios, without losing the delay-reduction benefits of deploying CoDel or PIE". The performance improvement is achieved when reacting to ECN-Echo in congestion avoidance (when ssthresh > cwnd) by multiplying cwnd and ssthresh with a value in the range [0.7,0.85]. Applying ABE when cwnd <= ssthresh is not currently recommended, but may benefit from additional attention, experimentation and specification.

4.1. Why Use ECN to Vary the Degree of Backoff?

AQM mechanisms such as CoDel [RFC8289] and PIE [RFC8033] set a delay target in routers and use congestion notifications to constrain the queuing delays experienced by packets, rather than in response to impending or actual bottleneck buffer exhaustion. With current default delay targets, CoDel and PIE both effectively emulate a bottleneck with a short queue (section II, [ABE2017]) while also allowing short traffic bursts into the queue. This provides acceptable performance for TCP connections over a path with a low BDP, or in highly multiplexed scenarios (many concurrent transport flows). However, in a lightly-multiplexed case over a path with a large BDP, conventional TCP backoff leads to gaps in packet transmission and under-utilisation of the path.

Instead of discarding packets, an AQM mechanism is allowed to mark ECN-Capable packets with an ECN CE-mark. The reception of a CE-mark feedback not only indicates congestion on the network path, it also indicates that an AQM mechanism exists at the bottleneck along the path, and hence the CE-mark likely came from a bottleneck with a controlled short queue. Reacting differently to an ECN-signalled congestion than to an inferred packet loss can then yield the benefit of a reduced back-off when queues are short. Using ECN can also be advantageous for several other reasons [RFC8087].

The idea of reacting differently to inferred packet loss and detection of an ECN-signalled congestion pre-dates this specification. For example, previous research proposed using ECN CE-marked feedback to modify TCP congestion control behaviour via a larger multiplicative decrease factor in conjunction with a smaller additive increase factor [ICC2002]. The goal of this former work was to operate across AQM bottlenecks using Random Early Detection (RED) that were not necessarily configured to emulate a short queue (The current usage of RED as an Internet AQM method is limited [RFC7567]).

4.2. An RTT-based response to indicated congestion

This specification applies to the use of ECN feedback as defined in [RFC3168], which specifies a response to indicated congestion that is no more frequent that once per path round trip time. Since ABE responds to indicated congestion once per RTT, it therefore does not respond to any further loss within the same RTT, because an ABE sender has already reduced the congestion window. If congestion persists after such reduction, ABE continues to reduce the congestion window in each consecutive RTT. This consecutive reduction can protect the network against long-standing unfairness in the case of AQM algorithms that do not keep a small average queue length. The mechanism does not rely on Accurate ECN ([I-D.ietf-tcpm-accurate-ecn]).

In contrast, transport protocol mechanisms can also be designed to utilise more frequent and detailed ECN feedback (e.g., Accurate ECN [I-D.ietf-tcpm-accurate-ecn]), which then permit a congestion control response that adjusts the sending rate more frequently. Datacenter TCP (DCTCP) [RFC8257] is an example of this approach.

5. ABE Deployment Requirements

This update is a sender-side only change. Like other changes to congestion control algorithms, it does not require any change to the TCP receiver or to network devices. It does not require any ABE-specific changes in routers or the use of Accurate ECN feedback [I-D.ietf-tcpm-accurate-ecn] by a receiver.

If the method is only deployed by some senders, and not by others, the senders that use this method can gain some advantage, possibly at the expense of other flows that do not use this updated method. Because this advantage applies only to ECN-marked packets and not to packet loss indications, an ECN-Capable bottleneck will still fall back to dropping packets if an TCP sender using ABE is too aggressive, and the result is no different than if the TCP sender was using traditional loss-based congestion control.

When used with bottlenecks that do not support ECN-marking the specification does not modify the transport protocol.

6. ABE Experiment Goals

[RFC3168] states that the congestion control response following an indication of ECN-signalled congestion is the same as the response to a dropped packet. [RFC8311] updates this specification to allow systems to provide a different behaviour when they experience ECN-signalled congestion rather than packet loss. The present specification defines such an experiment and has thus been assigned an Experimental status before being proposed as a Standards-Track update.

The purpose of the Internet experiment is to collect experience with deployment of ABE, and confirm acceptable safety in deployed networks that use this update to TCP congestion control. To evaluate ABE, this experiment therefore requires support in AQM routers for ECN-marking of packets carrying the ECN-Capable Transport, ECT(0), codepoint [RFC3168].

The result of this Internet experiment ought to include an investigation of the implications of experiencing an ECN-CE mark followed by loss within the same RTT. At the end of the experiment, this will be reported to the TCPM WG or the IESG.

7. Acknowledgements

Authors N. Khademi, M. Welzl and G. Fairhurst were part-funded by the European Community under its Seventh Framework Programme through the Reducing Internet Transport Latency (RITE) project (ICT-317700). The views expressed are solely those of the authors.

Author G. Armitage performed most of his work on this document while employed by Swinburne University of Technology, Melbourne, Australia.

The authors would like to thank Stuart Cheshire for many suggestions when revising the draft, and the following people for their contributions to [ABE2017]: Chamil Kulatunga, David Ros, Stein Gjessing, Sebastian Zander. Thanks also to (in alphabetical order) Roland Bless, Bob Briscoe, David Black, Markku Kojo, John Leslie, Lawrence Stewart, Dave Taht and the TCPM Working Group for providing valuable feedback on this document.

The authors would finally like to thank everyone who provided feedback on the congestion control behaviour specified in this update received from the IRTF Internet Congestion Control Research Group (ICCRG).

8. IANA Considerations

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

This document includes no request to IANA.

9. Implementation Status

ABE is implemented as a patch for Linux and FreeBSD. This is meant for research and available for download from http://heim.ifi.uio.no/michawe/research/abe/. This code was used to produce the test results that are reported in [ABE2017]. The FreeBSD code has been committed to the mainline kernel on March 19, 2018 [ABE-FreeBSD].

10. Security Considerations

The described method is a sender-side only transport change, and does not change the protocol messages exchanged. The security considerations for ECN [RFC3168] therefore still apply.

This is a change to TCP congestion control with ECN that will typically lead to a change in the capacity achieved when flows share a network bottleneck. This could result in some flows receiving more than their fair share of capacity. Similar unfairness in the way that capacity is shared is also exhibited by other congestion control mechanisms that have been in use in the Internet for many years (e.g., CUBIC [RFC8312]). Unfairness may also be a result of other factors, including the round trip time experienced by a flow. ABE applies only when ECN-marked packets are received, not when packets are lost, hence use of ABE cannot lead to congestion collapse.

11. Revision Information

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

-12. Corrections from Adam Roach; Benjamin Kaduk; & Ben Campbell

-10. Incorporated changes following the Gen-ART review by Russ Housley. Correction to URL.

-09. Changed to "Following publication of RFC 8311, this document specifies a sender-side change to TCP:"

-08. Addressed comments from AD review on the document structure, and relationship to existing RFCs.

-07. Addressed comments following WGLC.

- o Updated Reference citations.

- o Removed paragraph containing a wrong statement related to timeout in section 4.1.

- o Discuss what happens when cwnd <= ssthresh.

- o Added text on Concern about lower bound of 2*SMSS.

-06. Addressed Michael Scharf’s comments.

-05. Refined the description of the experiment based on feedback at IETF-100. Incorporated comments from David Black.

-04. Incorporates review comments from Lawrence Stewart and the remaining comments from Roland Bless. References are updated.

-03. Several review comments from Roland Bless are addressed. Consistent terminology and equations. Clarification on the scope of recommended beta_{ecn} value.

-02. Corrected the equations in Section 3.1. Updated the affiliations. Lower bound for cwnd is defined. A recommendation for window-based transport protocols is changed to cover all transport protocols that implement a congestion control reduction to an ECN congestion signal. Added text about ABE’s FreeBSD mainline kernel status including a reference to the FreeBSD code review page. References are updated.

-01. Text improved, mainly incorporating comments from Stuart Cheshire. The reference to a technical report has been updated to a published version of the tests [ABE2017]. Used "AQM Mechanism" throughout in place of other alternatives, and more consistent use of technical language and clarification on the intended purpose of the experiments required by EXP status. There was no change to the technical content.

-00. draft-ietf-tcpm-alternativebackoff-ecn-00 replaces draft-khademi-tcpm-alternativebackoff-ecn-01. Text describing the nature of the experiment was added.

Individual draft -01. This I-D now refers to draft-black-tsvwg-ecn-experimentation-02, which replaces draft-khademi-tsvwg-ecn-response-00 to make a broader update to RFC 3168 for the sake of allowing experiments. As a result, some of the motivating and discussing text that was moved from draft-khademi-alternativebackoff-ecn-03 to draft-khademi-tsvwg-ecn-response-00 has now been re-inserted here.

Individual draft -00. draft-khademi-tsvwg-ecn-response-00 and draft-khademi-tcpm-alternativebackoff-ecn-00 replace draft-khademi-alternativebackoff-ecn-03, following discussion in the TSVWG and TCPM working groups.

12. References

12.1. 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, <https://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, <https://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, <https://www.rfc-editor.org/info/rfc5681>.

[RFC7567] Baker, F., Ed. and G. Fairhurst, Ed., "IETF Recommendations Regarding Active Queue Management", BCP 197, RFC 7567, DOI 10.17487/RFC7567, July 2015, <https://www.rfc-editor.org/info/rfc7567>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8257] Bensley, S., Thaler, D., Balasubramanian, P., Eggert, L., and G. Judd, "Data Center TCP (DCTCP): TCP Congestion Control for Data Centers", RFC 8257, DOI 10.17487/RFC8257, October 2017, <https://www.rfc-editor.org/info/rfc8257>.

[RFC8311] Black, D., "Relaxing Restrictions on Explicit Congestion Notification (ECN) Experimentation", RFC 8311, DOI 10.17487/RFC8311, January 2018, <https://www.rfc-editor.org/info/rfc8311>.

12.2. Informative References

[ABE-FreeBSD] "ABE patch review in FreeBSD", <https://svnweb.freebsd.org/base?view=revision&revision=331214>.

[ABE2017] Khademi, N., Armitage, G., Welzl, M., Fairhurst, G., Zander, S., and D. Ros, "Alternative Backoff: Achieving Low Latency and High Throughput with ECN and AQM", IFIP NETWORKING 2017, Stockholm, Sweden, June 2017.

[BUFFERBLOAT]
Gettys, J. and K. Nichols, "Bufferbloat: Dark Buffers in the Internet", ACM Queue 9, 11, DOI 10.1145/2063166.2071893; https://queue.acm.org/detail.cfm?id=2071893", November 2011.

[CODEL2012]
Nichols, K. and V. Jacobson, "Controlling Queue Delay", July 2012, <http://queue.acm.org/detail.cfm?id=2209336>.

[I-D.ietf-tcpm-accurate-ecn]
Briscoe, B., Kuehlewind, M., and R. Scheffenegger, "More Accurate ECN Feedback in TCP", draft-ietf-tcpm-accurate-ecn-06 (work in progress), March 2018.

[ICC2002] Kwon, M. and S. Fahmy, "TCP Increase/Decrease Behavior with Explicit Congestion Notification (ECN)", IEEE ICC 2002, New York, New York, USA, May 2002, <http://dx.doi.org/10.1109/ICC.2002.997262>.

[RFC8033] Pan, R., Natarajan, P., Baker, F., and G. White, "Proportional Integral Controller Enhanced (PIE): A Lightweight Control Scheme to Address the Bufferbloat Problem", RFC 8033, DOI 10.17487/RFC8033, February 2017, <https://www.rfc-editor.org/info/rfc8033>.

[RFC8087] Fairhurst, G. and M. Welzl, "The Benefits of Using Explicit Congestion Notification (ECN)", RFC 8087, DOI 10.17487/RFC8087, March 2017, <https://www.rfc-editor.org/info/rfc8087>.

[RFC8289] Nichols, K., Jacobson, V., McGregor, A., Ed., and J. Iyengar, Ed., "Controlled Delay Active Queue Management", RFC 8289, DOI 10.17487/RFC8289, January 2018, <https://www.rfc-editor.org/info/rfc8289>.

[RFC8312] Rhee, I., Xu, L., Ha, S., Zimmermann, A., Eggert, L., and R. Scheffenegger, "CUBIC for Fast Long-Distance Networks", RFC 8312, DOI 10.17487/RFC8312, February 2018, <https://www.rfc-editor.org/info/rfc8312>.

Authors' Addresses

Naeem Khademi
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Email: naeemk@ifi.uio.no

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

Email: michawe@ifi.uio.no

Grenville Armitage
Netflix Inc.

Email: garmitage@netflix.com

Godred Fairhurst
University of Aberdeen
School of Engineering, Fraser Noble Building
Aberdeen AB24 3UE
UK

Email: gorry@erg.abdn.ac.uk

TCP Maintenance Working Group Y. Cheng
Internet-Draft N. Cardwell
Intended status: Experimental N. Dukkipati
Expires: September 6, 2018 P. Jha
Google, Inc
March 5, 2018

RACK: a time-based fast loss detection algorithm for TCP
draft-ietf-tcpm-rack-03

Abstract

This document presents a new TCP loss detection algorithm called RACK ("Recent ACKnowledgment"). RACK uses the notion of time, instead of packet or sequence counts, to detect losses, for modern TCP implementations that can support per-packet timestamps and the selective acknowledgment (SACK) option. It is intended to replace the conventional DUPACK threshold approach and its variants, as well as other nonstandard approaches.

Status of This Memo

Copyright Notice

1. Introduction

This document presents a new loss detection algorithm called RACK ("Recent ACKnowledgment"). RACK uses the notion of time instead of the conventional packet or sequence counting approaches for detecting losses. RACK deems a packet lost if some packet sent sufficiently later has been delivered. It does this by recording packet transmission times and inferring losses using cumulative acknowledgments or selective acknowledgment (SACK) TCP options.

In the last couple of years we have been observing several increasingly common loss and reordering patterns in the Internet:

1. Lost retransmissions. Traffic policers [POLICER16] and burst losses often cause retransmissions to be lost again, severely increasing TCP latency.

2. Tail drops. Structured request-response traffic turns more losses into tail drops. In such cases, TCP is application-limited, so it cannot send new data to probe losses and has to rely on retransmission timeouts (RTOs).

3. Reordering. Link layer protocols (e.g., 802.11 block ACK) or routers' internal load-balancing can deliver TCP packets out of order. The degree of such reordering is usually within the order of the path round trip time.

Despite TCP stacks (e.g. Linux) that implement many of the standard and proposed loss detection algorithms [RFC3517][RFC4653][RFC5827][RFC5681][RFC6675][RFC7765][FACK][THIN-STREAM][TLP], we've found that together they do not perform well. The main reason is that many of them are based on the classic rule of counting duplicate acknowledgments [RFC5681]. They can either detect loss quickly or accurately, but not both, especially when the sender is application-limited or under reordering that is unpredictable. And under these conditions none of them can detect lost retransmissions well.

Also, these algorithms, including RFCs, rarely address the interactions with other algorithms. For example, FACK may consider a packet is lost while RFC3517 may not. Implementing N algorithms while dealing with N^2 interactions is a daunting task and error-prone.

The goal of RACK is to solve all the problems above by replacing many of the loss detection algorithms above with one simpler, and also more effective, algorithm.

2. Overview

The main idea behind RACK is that if a packet has been delivered out of order, then the packets sent chronologically before that were either lost or reordered. This concept is not fundamentally different from [RFC5681][RFC3517][FACK]. But the key innovation in RACK is to use a per-packet transmission timestamp and widely deployed SACK options to conduct time-based inferences instead of inferring losses with packet or sequence counting approaches.

Using a threshold for counting duplicate acknowledgments (i.e., DupThresh) is no longer reliable because of today’s prevalent reordering patterns. A common type of reordering is that the last "runt" packet of a window’s worth of packet bursts gets delivered first, then the rest arrive shortly after in order. To handle this effectively, a sender would need to constantly adjust the DupThresh to the burst size; but this would risk increasing the frequency of RTOs on real losses.

Today’s prevalent lost retransmissions also cause problems with packet-counting approaches [RFC5681][RFC3517][FACK], since those approaches depend on reasoning in sequence number space. Retransmissions break the direct correspondence between ordering in sequence space and ordering in time. So when retransmissions are lost, sequence-based approaches are often unable to infer and quickly repair losses that can be deduced with time-based approaches.

Instead of counting packets, RACK uses the most recently delivered packet’s transmission time to judge if some packets sent previous to that time have "expired" by passing a certain reordering settling window. On each ACK, RACK marks any already-expired packets lost, and for any packets that have not yet expired it waits until the reordering window passes and then marks those lost as well. In either case, RACK can repair the loss without waiting for a (long) RTO. RACK can be applied to both fast recovery and timeout recovery, and can detect losses on both originally transmitted and retransmitted packets, making it a great all-weather loss detection mechanism.

3. Requirements

The reader is expected to be familiar with the definitions given in the TCP congestion control [RFC5681] and selective acknowledgment

[RFC2018] RFCs. Familiarity with the conservative SACK-based recovery for TCP [RFC6675] is not expected but helps.

RACK has three requirements:

1. The connection MUST use selective acknowledgment (SACK) options [RFC2018].

2. For each packet sent, the sender MUST store its most recent transmission time with (at least) millisecond granularity. For round-trip times lower than a millisecond (e.g., intra-datacenter communications) microsecond granularity would significantly help the detection latency but is not required.

3. For each packet sent, the sender MUST remember whether the packet has been retransmitted or not.

We assume that requirement 1 implies the sender keeps a SACK scoreboard, which is a data structure to store selective acknowledgment information on a per-connection basis ([RFC6675] section 3). For the ease of explaining the algorithm, we use a pseudo-scoreboard that manages the data in sequence number ranges. But the specifics of the data structure are left to the implementor.

RACK does not need any change on the receiver.

4. Definitions of variables

A sender needs to store these new RACK variables:

"Packet.xmit_ts" is the time of the last transmission of a data packet, including retransmissions, if any. The sender needs to record the transmission time for each packet sent and not yet acknowledged. The time MUST be stored at millisecond granularity or finer.

"RACK.packet". Among all the packets that have been either selectively or cumulatively acknowledged, RACK.packet is the one that was sent most recently (including retransmissions).

"RACK.xmit_ts" is the latest transmission timestamp of RACK.packet.

"RACK.end_seq" is the ending TCP sequence number of RACK.packet.

"RACK.RTT" is the associated RTT measured when RACK.xmit_ts, above, was changed. It is the RTT of the most recently transmitted packet that has been delivered (either cumulatively acknowledged or selectively acknowledged) on the connection.

"RACK.reo_wnd" is a reordering window for the connection, computed in the unit of time used for recording packet transmission times. It is used to defer the moment at which RACK marks a packet lost.

"RACK.min_RTT" is the estimated minimum round-trip time (RTT) of the connection.

"RACK.ack_ts" is the time when all the sequences in RACK.packet were selectively or cumulatively acknowledged.

"RACK.reo_wnd_incr" is the multiplier applied to adjust RACK.reo_wnd

"RACK.reo_wnd_persist" is the number of loss recoveries before resetting RACK.reo_wnd "RACK.dsack" indicates if RACK.reo_wnd has been adjusted upon receiving a DSACK option

Note that the Packet.xmit_ts variable is per packet in flight. The RACK.xmit_ts, RACK.end_seq, RACK.RTT, RACK.reo_wnd, and RACK.min_RTT variables are kept in the per-connection TCP control block. RACK.packet and RACK.ack_ts are used as local variables in the algorithm.

5. Algorithm Details

5.1. Transmitting a data packet

Upon transmitting a new packet or retransmitting an old packet, record the time in Packet.xmit_ts. RACK does not care if the retransmission is triggered by an ACK, new application data, an RTO, or any other means.

5.2. Upon receiving an ACK

Step 1: Update RACK.min_RTT.

Use the RTT measurements obtained via [RFC6298] or [RFC7323] to update the estimated minimum RTT in RACK.min_RTT. The sender can track a simple global minimum of all RTT measurements from the connection, or a windowed min-filtered value of recent RTT measurements. This document does not specify an exact approach.

Step 2: Update RACK stats

Given the information provided in an ACK, each packet cumulatively ACKed or SACKed is marked as delivered in the scoreboard. Among all the packets newly ACKed or SACKed in the connection, record the most recent Packet.xmit_ts in RACK.xmit_ts if it is ahead of RACK.xmit_ts. Sometimes the timestamps of RACK.Packet and Packet could carry the same transmit timestamps due to clock granularity or segmentation offloading (i.e. the two packets were sent as a jumbo frame into the NIC). In that case the sequence numbers of RACK.end_seq and Packet.end_seq are compared to break the tie.

Since an ACK can also acknowledge retransmitted data packets, RACK.RTT can be vastly underestimated if the retransmission was spurious. To avoid that, ignore a packet if any of its TCP sequences have been retransmitted before and either of two conditions is true:

1. The Timestamp Echo Reply field (TSecr) of the ACK’s timestamp option [RFC7323], if available, indicates the ACK was not acknowledging the last retransmission of the packet.

2. The packet was last retransmitted less than RACK.min_rtt ago. While it is still possible the packet is spuriously retransmitted because of a recent RTT decrease, we believe that our experience suggests this is a reasonable heuristic.

If the ACK is not ignored as invalid, update the RACK.RTT to be the RTT sample calculated using this ACK, and continue. If this ACK or SACK was for the most recently sent packet, then record the RACK.xmit_ts timestamp and RACK.end_seq sequence implied by this ACK. Otherwise exit here and omit the following steps.

Step 2 may be summarized in pseudocode as:

```
RACK_sent_after(t1, seq1, t2, seq2):
    If t1 > t2:
        Return true
    Else if t1 == t2 AND seq1 > seq2:
        Return true
    Else:
        Return false

RACK_update():
    For each Packet newly acknowledged cumulatively or selectively:
        rtt = Now() - RACK.xmit_ts
        If Packet has been retransmitted:
            If ACK.ts_option.echo_reply < Packet.xmit_ts:
               Return
            If rtt < RACK.min_rtt:
               Return

        RACK.RTT = rtt
        If RACK_sent_after(Packet.xmit_ts, Packet.end_seq
                           RACK.xmit_ts, RACK.end_seq):
            RACK.xmit_ts = Packet.xmit_ts
            RACK.end_seq = Packet.end_seq
```

Step 3: Update RACK reordering window

To handle the prevalent small degree of reordering, RACK.reo_wnd serves as an allowance for settling time before marking a packet lost. Use a conservative window of min_RTT / 4 if the connection is not currently in loss recovery. When in loss recovery, use a RACK.reo_wnd of zero in order to retransmit quickly.

Extension 1: Optionally size the window based on DSACK Further, the sender MAY leverage DSACK [RFC3708] to adapt the reordering window to higher degrees of reordering. Receiving an ACK with a DSACK indicates a spurious retransmission, which in turn suggests that the RACK reordering window, RACK.reo_wnd, is likely too small. The sender MAY increase the RACK.reo_wnd window linearly for every round trip in which the sender receives a DSACK, so that after N distinct round trips in which a DSACK is received, the RACK.reo_wnd is N * min_RTT / 4. The inflated RACK.reo_wnd would persist for 16 loss recoveries and then reset to its starting value, min_RTT / 4.

Extension 2: Optionally size the window if reordering has been observed

If the reordering window is too small or the connection does not support DSACK, then RACK can trigger spurious loss recoveries and reduce the congestion window unnecessarily. If the implementation supports reordering detection such as [REORDER-DETECT], then the sender MAY use the dynamically-sized reordering window based on min_RTT during loss recovery instead of a zero reordering window to compensate. Extension 3: Optionally size the window with the classic DUPACK threshold heuristic The DUPACK threshold approach in the current standards [RFC5681][RFC6675] is simple, and for decades has been effective in quickly detecting losses, despite the drawbacks discussed earlier. RACK can easily maintain the DUPACK threshold's advantages of quick detection by resetting the reordering window to zero (using RACK.reo_wnd = 0) when the DUPACK threshold is met (i.e. when at least three packets have been selectively acknowledged). The subtle differences are discussed in the section "RACK and TLP discussions".

The following algorithm includes the basic and all the extensions mentioned above. Note that individual extensions that require additional TCP features (e.g. DSACK) would work if the feature functions simply return false.

```
RACK_update_reo_wnd:
    RACK.min_RTT = TCP_min_RTT()
    If RACK_ext_TCP_ACK_has_DSACK_option():
        RACK.dsack = true

    If SND.UNA < RACK.roundtrip_seq:
        RACK.dsack = false  /* React to DSACK once within a round trip */

    If RACK.dsack:
        RACK.reo_wnd_incr += 1
        RACK.dsack = false
        RACK.roundtrip_seq = SND.NXT
        RACK.reo_wnd_persist = 16 /* Keep window for 16 loss recoveries */
    Else if exiting loss recovery:
        RACK.reo_wnd_persist -= 1
        If RACK.reo_wnd_persist <= 0:
            RACK.reo_wnd_incr = 1

    If in loss recovery and not RACK_ext_TCP_seen_reordering():
        RACK.reo_wnd = 0
    Else if RACK_ext_TCP_dupack_threshold_hit(): /* DUPTHRESH emulation mode */
        RACK.reo_wnd = 0
    Else:
        RACK.reo_wnd = RACK.min_RTT / 4 * RACK.reo_wnd_incr
        RACK.reo_wnd = min(RACK.reo_wnd, SRTT)
```

Step 4: Detect losses.

For each packet that has not been SACKed, if RACK.xmit_ts is after Packet.xmit_ts + RACK.reo_wnd, then mark the packet (or its corresponding sequence range) lost in the scoreboard. The rationale is that if another packet that was sent later has been delivered, and the reordering window or "reordering settling time" has already passed, then the packet was likely lost.

If another packet that was sent later has been delivered, but the reordering window has not passed, then it is not yet safe to deem the unacked packet lost. Using the basic algorithm above, the sender would wait for the next ACK to further advance RACK.xmit_ts; but this risks a timeout (RTO) if no more ACKs come back (e.g, due to losses or application limit). For timely loss detection, the sender MAY install a "reordering settling" timer set to fire at the earliest moment at which it is safe to conclude that some packet is lost. The earliest moment is the time it takes to expire the reordering window of the earliest unacked packet in flight.

This timer expiration value can be derived as follows. As a starting point, we consider that the reordering window has passed if the RACK.packet was sent sufficiently after the packet in question, or a sufficient time has elapsed since the RACK.packet was S/ACKed, or some combination of the two. More precisely, RACK marks a packet as lost if the reordering window for a packet has elapsed through the sum of:

1. delta in transmit time between a packet and the RACK.packet

2. delta in time between RACK.ack_ts and now

So we mark a packet as lost if:

```
RACK.xmit_ts >= Packet.xmit_ts
        AND
(RACK.xmit_ts - Packet.xmit_ts) + (now - RACK.ack_ts) >= RACK.reo_wnd
```

If we solve this second condition for "now", the moment at which we can declare a packet lost, then we get:

```
now >= Packet.xmit_ts + RACK.reo_wnd + (RACK.ack_ts - RACK.xmit_ts)
```

Then (RACK.ack_ts - RACK.xmit_ts) is just the RTT of the packet we used to set RACK.xmit_ts, so this reduces to:

```
Packet.xmit_ts + RACK.RTT + RACK.reo_wnd - now <= 0
```

The following pseudocode implements the algorithm above. When an ACK is received or the RACK timer expires, call RACK_detect_loss(). The algorithm includes an additional optimization to break timestamp ties by using the TCP sequence space. The optimization is particularly useful to detect losses in a timely manner with TCP Segmentation Offload, where multiple packets in one TSO blob have identical timestamps. It is also useful when the timestamp clock granularity is close to or longer than the actual round trip time.

```
RACK_detect_loss():
    timeout = 0

    For each packet, Packet, in the scoreboard:
        If Packet is already SACKed
            or marked lost and not yet retransmitted:
            Continue

        If RACK_sent_after(RACK.xmit_ts, RACK.end_seq,
                           Packet.xmit_ts, Packet.end_seq):
            remaining = Packet.xmit_ts + RACK.RTT + RACK.reo_wnd - Now()
            If remaining <= 0:
                Mark Packet lost
            Else:
                timeout = max(remaining, timeout)

    If timeout != 0
        Arm a timer to call RACK_detect_loss() after timeout
```

Implementation optimization: looping through packets in the SACK scoreboard above could be very costly on large BDP networks since the inflight could be very large. If the implementation can organize the scoreboard data structures to have packets sorted by the last (re)transmission time, then the loop can start on the least recently sent packet and aborts on the first packet sent after RACK.time_ts. This can be implemented by using a seperate list sorted in time order. The implementation inserts the packet to the tail of the list when it is (re)transmitted, and removes a packet from the list when it is delivered or marked lost. We RECOMMEND such an optimization for implementations for support high BDP networks. The optimization is implemented in Linux and sees orders of magnitude improvement on CPU usage on high speed WAN networks.

Tail Loss Probe: fast recovery on tail losses

This section describes a supplemental algorithm, Tail Loss Probe (TLP), which leverages RACK to further reduce RTO recoveries. TLP triggers fast recovery to quickly repair tail losses that can otherwise be recovered by RTOs only. After an original data transmission, TLP sends a probe data segment within one to two RTTs. The probe data segment can either be new, previously unsent data, or a retransmission of previously sent data just below SND.NXT. In either case the goal is to elicit more feedback from the receiver, in the form of an ACK (potentially with SACK blocks), to allow RACK to trigger fast recovery instead of an RTO.

An RTO occurs when the first unacknowledged sequence number is not acknowledged after a conservative period of time has elapsed [RFC6298]. Common causes of RTOs include:

1. The entire flight is lost

2. Tail losses at the end of an application transaction

3. Lost retransmits, which can halt fast recovery based on [RFC6675] if the ACK stream completely dries up. For example, consider a window of three data packets (P1, P2, P3) that are sent; P1 and P2 are dropped. On receipt of a SACK for P3, RACK marks P1 and P2 as lost and retransmits them as R1 and R2. Suppose R1 and R2 are lost as well, so there are no more returning ACKs to detect R1 and R2 as lost. Recovery stalls.

4. Tail losses of ACKs.

5. An unexpectedly long round-trip time (RTT). This can cause ACKs to 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, but F-RTO cannot be used in many situations.

5.3. Tail Loss Probe: An Example

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. Sender receives acknowledgements (ACKs) for segments 1-5; segments 6-10 are lost and no ACKs are received. The sender reschedules 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 (2) of the algorithm.

3. When PTO fires, sender retransmits segment 10.

4. After an RTT, a SACK for packet 10 arrives. The ACK also carries SACK holes for segments 6, 7, 8 and 9. This triggers RACK-based loss recovery.

5. The connection enters fast recovery and retransmits the remaining lost segments.

5.4. Tail Loss Probe Algorithm Details

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 (PTO) 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 as specified in [RFC6298].

Open state: the sender's loss recovery state machine is in its normal, default state: there are no SACKed sequence ranges in the SACK scoreboard, and neither fast recovery, timeout-based recovery, nor ECN-based cwnd reduction are underway.

The TLP algorithm has three phases, which we discuss in turn.

5.4.1. Phase 1: Scheduling a loss probe

Step 1: Check conditions for scheduling a PTO.

A sender should check to see if it should schedule a PTO in two situations:

1. After transmitting new data

2. Upon receiving an ACK that cumulatively acknowledges data.

A sender should schedule a PTO only if all of the following conditions are met:

1. The connection supports SACK [RFC2018]

2. The connection is not in loss recovery

3. The connection is either limited by congestion window (the data in flight matches or exceeds the cwnd) or application-limited (there is no unsent data that the receiver window allows to be sent).

4. The most recently transmitted data was not itself a TLP probe (i.e. a sender MUST NOT send consecutive or back-to-back TLP probes).

If a PTO cannot be scheduled according to these conditions, then the sender MUST arm the RTO timer if there is unacknowledged data in flight.

Step 2: Select the duration of the PTO.

A sender SHOULD use the following logic to select the duration of a PTO:

```
TLP_timeout():
    If SRTT is available:
        PTO = 2 * SRTT
        If FlightSize = 1:
           PTO += WCDelAckT
        Else:
           PTO += 2ms
    Else:
        PTO = 1 sec

    If Now() + PTO > TCP_RTO_expire():
        PTO = TCP_RTO_expire() - Now()
```

Aiming for a PTO value of 2*SRTT allows a sender to wait long enough to know that an ACK is overdue. Under normal circumstances, i.e. no losses, an ACK typically arrives in one SRTT. But choosing PTO to be exactly an SRTT is likely to generate spurious probes given that network delay variance and even end-system timings can easily push an ACK to be above an SRTT. We chose PTO to be the next integral multiple of SRTT.

Similarly, current end-system processing latencies and timer granularities can easily delay ACKs, so senders SHOULD add at least 2ms to a computed PTO value (and MAY add more if the sending host OS timer granularity is more coarse than 1ms).

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

Finally, if the time at which an RTO would fire (here denoted "TCP_RTO_expire") is sooner than the computed time for the PTO, then a probe is scheduled to be sent at that earlier time..

5.4.2. Phase 2: Sending a loss probe

When the PTO fires, transmit a probe data segment:

```
TLP_send_probe():
    If a previously unsent segment exists AND
       the receive window allows new data to be sent:
        Transmit that new segment
        FlightSize += SMSS
    Else:
        Retransmit the last segment
    The cwnd remains unchanged
```

5.4.3. Phase 3: ACK processing

On each incoming ACK, the sender should cancel any existing loss probe timer. The sender should then reschedule the loss probe timer if the conditions in Step 1 of Phase 1 allow.

5.5. TLP recovery detection

If the only loss in an outstanding window of data was the last segment, then a TLP loss probe retransmission of that data segment might repair the loss. TLP recovery detection examines ACKs to detect when the probe might have repaired a loss, and thus allows congestion control to properly reduce the congestion window (cwnd) [RFC5681].

Consider a TLP retransmission episode where a sender retransmits a tail packet in a flight. The TLP retransmission episode ends when the sender receives an ACK with a SEG.ACK above the SND.NXT at the time the episode started. During the TLP retransmission episode the sender checks for a duplicate ACK or D-SACK indicating that both the original segment and TLP retransmission arrived at the receiver, meaning there was no loss that needed repairing. If the TLP sender does not receive such an indication before the end of the TLP retransmission episode, then it MUST estimate that either the original data segment or the TLP retransmission were lost, and congestion control MUST react appropriately to that loss as it would any other loss.

Since 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 SACK support [RFC2018].

Definitions of variables

TLPRxtOut: a boolean indicating whether there is an unacknowledged TLP retransmission.

TLPHighRxt: the value of SND.NXT at the time of sending a TLP retransmission.

5.5.1. Initializing and resetting state

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

    TLPRxtOut = false

5.5.2. Recording loss probe states

Senders must only send a TLP loss probe retransmission if TLPRxtOut is false. This ensures that at any given time a connection has at most one outstanding TLP retransmission. This allows the sender to use the algorithm described in this section to estimate whether any data segments were lost.

Note that this condition only restricts 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 retransmission timeout and fast recovery algorithms are sufficient to detect losses of such probe segments.

Upon sending a TLP probe that is a retransmission, the sender sets TLPRxtOut to true and TLPHighRxt to SND.NXT.

Detecting recoveries accomplished by loss probes

Step 1: Track ACKs indicating receipt of original and retransmitted segments

A sender considers both the original segment and TLP probe retransmission segment as acknowledged if either 1 or 2 are true:

1. This is a duplicate acknowledgment (as defined in [RFC5681], section 2), and all of the following conditions are met:

    1. TLPRxtOut is true

2. SEG.ACK == TLPHighRxt

3. SEG.ACK == SND.UNA

4. the segment contains no SACK blocks for sequence ranges above TLPHighRxt

5. the segment contains no data

6. the segment is not a window update

2. This is an ACK acknowledging a sequence number at or above TLPHighRxt and it contains a D-SACK; i.e. all of the following conditions are met:

1. TLPRxtOut is true

2. SEG.ACK >= TLPHighRxt

3. the ACK contains a D-SACK block

If neither conditions are met, then the sender estimates that the receiver received both the original data segment and the TLP probe retransmission, and so the sender considers the TLP episode to be done, and records that fact by setting TLPRxtOut to false.

Step 2: Mark the end of a TLP retransmission episode and detect losses

If the sender receives a cumulative 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 for the original segment and TLP probe retransmission segment. At that time, if the TLPRxtOut flag is still true and thus indicates that the TLP probe retransmission remains unacknowledged, then the sender should presume that at least one of its data segments was lost, so it SHOULD invoke a congestion control response equivalent to fast recovery.

More precisely, on each ACK the sender executes the following:

```
if (TLPRxtOut and SEG.ACK >= TLPHighRxt) {
    TLPRxtOut = false
    EnterRecovery()
    ExitRecovery()
}
```

6. RACK and TLP discussions

6.1. Advantages

The biggest advantage of RACK is that every data packet, whether it is an original data transmission or a retransmission, can be used to detect losses of the packets sent chronologically prior to it.

Example: TAIL DROP. Consider a sender that transmits a window of three data packets (P1, P2, P3), and P1 and P3 are lost. Suppose the transmission of each packet is at least RACK.reo_wnd (1 millisecond by default) after the transmission of the previous packet. RACK will mark P1 as lost when the SACK of P2 is received, and this will trigger the retransmission of P1 as R1. When R1 is cumulatively acknowledged, RACK will mark P3 as lost and the sender will retransmit P3 as R3. This example illustrates how RACK is able to repair certain drops at the tail of a transaction without any timer. Notice that neither the conventional duplicate ACK threshold [RFC5681], nor [RFC6675], nor the Forward Acknowledgment [FACK] algorithm can detect such losses, because of the required packet or sequence count.

Example: LOST RETRANSMIT. Consider a window of three data packets (P1, P2, P3) that are sent; P1 and P2 are dropped. Suppose the transmission of each packet is at least RACK.reo_wnd (1 millisecond by default) after the transmission of the previous packet. When P3 is SACKed, RACK will mark P1 and P2 lost and they will be retransmitted as R1 and R2. Suppose R1 is lost again but R2 is SACKed; RACK will mark R1 lost for retransmission again. Again, neither the conventional three duplicate ACK threshold approach, nor [RFC6675], nor the Forward Acknowledgment [FACK] algorithm can detect such losses. And such a lost retransmission is very common when TCP is being rate-limited, particularly by token bucket policers with large bucket depth and low rate limit. Retransmissions are often lost repeatedly because standard congestion control requires multiple round trips to reduce the rate below the policed rate.

Example: SMALL DEGREE OF REORDERING. Consider a common reordering event: a window of packets are sent as (P1, P2, P3). P1 and P2 carry a full payload of MSS octets, but P3 has only a 1-octet payload. Suppose the sender has detected reordering previously (e.g., by implementing the algorithm in [REORDER-DETECT]) and thus RACK.reo_wnd is min_RTT/4. Now P3 is reordered and delivered first, before P1 and P2. As long as P1 and P2 are delivered within min_RTT/4, RACK will not consider P1 and P2 lost. But if P1 and P2 are delivered outside the reordering window, then RACK will still falsely mark P1 and P2 lost. We discuss how to reduce false positives in the end of this section.

The examples above show that RACK is particularly useful when the sender is limited by the application, which is common for interactive, request/response traffic. Similarly, RACK still works when the sender is limited by the receive window, which is common for applications that use the receive window to throttle the sender.

For some implementations (e.g., Linux), RACK works quite efficiently with TCP Segmentation Offload (TSO). RACK always marks the entire TSO blob lost because the packets in the same TSO blob have the same transmission timestamp. By contrast, the counting based algorithms (e.g., [RFC3517][RFC5681]) may mark only a subset of packets in the TSO blob lost, forcing the stack to perform expensive fragmentation of the TSO blob, or to selectively tag individual packets lost in the scoreboard.

6.2. Disadvantages

RACK requires the sender to record the transmission time of each packet sent at a clock granularity of one millisecond or finer. TCP implementations that record this already for RTT estimation do not require any new per-packet state. But implementations that are not yet recording packet transmission times will need to add per-packet internal state (commonly either 4 or 8 octets per packet or TSO blob) to track transmission times. In contrast, the conventional [RFC6675] loss detection approach does not require any per-packet state beyond the SACK scoreboard. This is particularly useful on ultra-low RTT networks where the RTT is far less than the sender TCP clock grainularity (e.g. inside data-centers).

RACK can easily and optionally support the conventional approach in [RFC6675][RFC5681] by resetting the reordering window to zero when the threshold is met. Note that this approach differs slightly from [RFC6675] which considers a packet lost when at least #DupThresh higher-sequenc packets are SACKed. RACK's approach considers a packet lost when at least one higher sequence packet is SACKed and the total number of SACKed packets is at least DupThresh. For example, suppose a connection sends 10 packets, and packets 3, 5, 7 are SACKed. [RFC6675] considers packets 1 and 2 lost. RACK considers packets 1, 2, 4, 6 lost.

6.3. Adjusting the reordering window

When the sender detects packet reordering, RACK uses a reordering window of min_rtt / 4. It uses the minimum RTT to accommodate reordering introduced by packets traversing slightly different paths (e.g., router-based parallelism schemes) or out-of-order deliveries in the lower link layer (e.g., wireless links using link-layer retransmission). RACK uses a quarter of minimum RTT because Linux

TCP used the same factor in its implementation to delay Early Retransmit [RFC5827] to reduce spurious loss detections in the presence of reordering, and experience shows that this seems to work reasonably well. We have evaluated using the smoothed RTT (SRTT from [RFC6298] RTT estimation) or the most recently measured RTT (RACK.RTT) using an experiment similar to that in the Performance Evaluation section. They do not make any significant difference in terms of total recovery latency.

6.4. Relationships with other loss recovery algorithms

The primary motivation of RACK is to ultimately provide a simple and general replacement for some of the standard loss recovery algorithms [RFC5681][RFC6675][RFC5827][RFC4653], as well as some nonstandard ones [FACK][THIN-STREAM]. While RACK can be a supplemental loss detection mechanism on top of these algorithms, this is not necessary, because RACK implicitly subsumes most of them.

[RFC5827][RFC4653][THIN-STREAM] dynamically adjusts the duplicate ACK threshold based on the current or previous flight sizes. RACK takes a different approach, by using only one ACK event and a reordering window. RACK can be seen as an extended Early Retransmit [RFC5827] without a FlightSize limit but with an additional reordering window. [FACK] considers an original packet to be lost when its sequence range is sufficiently far below the highest SACKed sequence. In some sense RACK can be seen as a generalized form of FACK that operates in time space instead of sequence space, enabling it to better handle reordering, application-limited traffic, and lost retransmissions.

Nevertheless RACK is still an experimental algorithm. Since the oldest loss detection algorithm, the 3 duplicate ACK threshold [RFC5681], has been standardized and widely deployed. RACK can easily and optionally support the conventional approach for compatibility.

RACK is compatible with and does not interfere with the the standard RTO [RFC6298], RTO-restart [RFC7765], F-RTO [RFC5682] and Eifel algorithms [RFC3522]. This is because RACK only detects loss by using ACK events. It neither changes the RTO timer calculation nor detects spurious timeouts.

Furthermore, RACK naturally works well with Tail Loss Probe [TLP] because a tail loss probe solicits either an ACK or SACK, which can be used by RACK to detect more losses. RACK can be used to relax TLP's requirement for using FACK and retransmitting the the highest-sequenced packet, because RACK is agnostic to packet sequence numbers, and uses transmission time instead. Thus TLP could be modified to retransmit the first unacknowledged packet, which could improve application latency.

6.5. Interaction with congestion control

RACK intentionally decouples loss detection from congestion control. RACK only detects losses; it does not modify the congestion control algorithm [RFC5681][RFC6937]. However, RACK may detect losses earlier or later than the conventional duplicate ACK threshold approach does. A packet marked lost by RACK SHOULD NOT be retransmitted until congestion control deems this appropriate. Specifically, Proportional Rate Reduction [RFC6937] SHOULD be used when using RACK.

RACK is applicable for both fast recovery and recovery after a retransmission timeout (RTO) in [RFC5681]. RACK applies equally to fast recovery and RTO recovery because RACK is purely based on the transmission time order of packets. When a packet retransmitted by RTO is acknowledged, RACK will mark any unacked packet sent sufficiently prior to the RTO as lost, because at least one RTT has elapsed since these packets were sent.

The following simple example compares how RACK and non-RACK loss detection interacts with congestion control: suppose a TCP sender has a congestion window (cwnd) of 20 packets on a SACK-enabled connection. It sends 10 data packets and all of them are lost.

Without RACK, the sender would time out, reset cwnd to 1, and retransmit the first packet. It would take four round trips (1 + 2 + 4 + 3 = 10) to retransmit all the 10 lost packets using slow start. The recovery latency would be RTO + 4*RTT, with an ending cwnd of 4 packets due to congestion window validation.

With RACK, a sender would send the TLP after 2*RTT and get a DUPACK. If the sender implements Proportional Rate Reduction [RFC6937] it would slow start to retransmit the remaining 9 lost packets since the number of packets in flight (0) is lower than the slow start threshold (10). The slow start would again take four round trips (1 + 2 + 4 + 3 = 10). The recovery latency would be 2*RTT + 4*RTT, with an ending cwnd set to the slow start threshold of 10 packets.

In both cases, the sender after the recovery would be in congestion avoidance. The difference in recovery latency (RTO + 4*RTT vs 6*RTT) can be significant if the RTT is much smaller than the minimum RTO (1 second in RFC6298) or if the RTT is large. The former case is common in local area networks, data-center networks, or content distribution networks with deep deployments. The latter case is more common in developing regions with highly congested and/or high-latency networks. The ending congestion window after recovery also impacts subsequent data transfer.

6.6. TLP recovery detection with delayed ACKs

Delayed ACKs complicate the detection of repairs done by TLP, since with a delayed ACK 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 above features such a delay, in the form of WCDelAckT. Furthermore, if the receiver supports duplicate selective acknowledgments (D-SACKs) [RFC2883] then in the case of a delayed ACK the sender's TLP recovery detection algorithm (see above) can use the D-SACK information to infer that the original and TLP retransmission both arrived at the receiver.

If there is ACK loss or a delayed ACK without a D-SACK, 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 can be prudent.

6.7. RACK for other transport protocols

RACK can be implemented in other transport protocols. The algorithm can be simplified by skipping step 3 if the protocol can support a unique transmission or packet identifier (e.g. TCP echo options). For example, the QUIC protocol implements RACK [QUIC-LR].

7. Experiments and Performance Evaluations

RACK and TLP have been deployed at Google, for both connections to users in the Internet and internally. We conducted a performance evaluation experiment for RACK and TLP on a small set of Google Web servers in Western Europe that serve mostly European and some African countries. The experiment lasted three days in March 2017. The servers were divided evenly into four groups of roughly 5.3 million flows each:

Group 1 (control): RACK off, TLP off, RFC 3517 on

Group 2: RACK on, TLP off, RFC 3517 on

Group 3: RACK on, TLP on, RFC 3517 on

Group 4: RACK on, TLP on, RFC 3517 off

All groups used Linux with CUBIC congestion control, an initial congestion window of 10 packets, and the fq/pacing qdisc. In terms of specific recovery features, all groups enabled RFC5682 (F-RTO) but disabled FACK because it is not an IETF RFC. FACK was excluded because the goal of this setup is to compare RACK and TLP to RFC-based loss recoveries. Since TLP depends on either FACK or RACK, we could not run another group that enables TLP only (with both RACK and FACK disabled). Group 4 is to test whether RACK plus TLP can completely replace the DupThresh-based [RFC3517].

The servers sit behind a load balancer that distributes the connections evenly across the four groups.

Each group handles a similar number of connections and sends and receives similar amounts of data. We compare total time spent in loss recovery across groups. The recovery time is measured from when the recovery and retransmission starts, until the remote host has acknowledged the highest sequence (SND.NXT) at the time the recovery started. Therefore the recovery includes both fast recoveries and timeout recoveries.

Our data shows that Group 2 recovery latency is only 0.3% lower than the Group 1 recovery latency. But Group 3 recovery latency is 25% lower than Group 1 due to a 40% reduction in RTO-triggered recoveries! Therefore it is important to implement both TLP and RACK for performance. Group 4’s total recovery latency is 0.02% lower than Group 3’s, indicating that RACK plus TLP can successfully replace RFC3517 as a standalone recovery mechanism.

We want to emphasize that the current experiment is limited in terms of network coverage. The connectivity in Western Europe is fairly good, therefore loss recovery is not a major performance bottleneck. We plan to expand our experiments to regions with worse connectivity, in particular on networks with strong traffic policing.

8. Security Considerations

RACK does not change the risk profile for TCP.

An interesting scenario is ACK-splitting attacks [SCWA99]: for an MSS-size packet sent, the receiver or the attacker might send MSS ACKs that SACK or acknowledge one additional byte per ACK. This would not fool RACK. RACK.xmit_ts would not advance because all the sequences of the packet are transmitted at the same time (carry the same transmission timestamp). In other words, SACKing only one byte of a packet or SACKing the packet in entirety have the same effect on RACK.

9. IANA Considerations

This document makes no request of IANA.

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

10. Acknowledgments

The authors thank Matt Mathis for his insights in FACK and Michael Welzl for his per-packet timer idea that inspired this work. Eric Dumazet, Randy Stewart, Van Jacobson, Ian Swett, Rick Jones, Jana Iyengar, and Hiren Panchasara contributed to the draft and the implementations in Linux, FreeBSD and QUIC.

11. References

11.1. Normative References

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

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

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

[RFC4737] Morton, A., Ciavattone, L., Ramachandran, G., Shalunov, S., and J. Perser, "Packet Reordering Metrics", RFC 4737, November 2006.

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

[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.

[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.

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

[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.

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

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

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

11.2. Informative References

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

[POLICER16] Flach, T., Papageorge, P., Terzis, A., Pedrosa, L., Cheng, Y., Karim, T., Katz-Bassett, E., and R. Govindan, "An Analysis of Traffic Policing in the Web", ACM SIGCOMM , 2016.

[QUIC-LR] Iyengar, J. and I. Swett, "QUIC Loss Recovery And Congestion Control", draft-tsvwg-quic-loss-recovery-01 (work in progress), June 2016.

[REORDER-DETECT] Zimmermann, A., Schulte, L., Wolff, C., and A. Hannemann, "Detection and Quantification of Packet Reordering with TCP", draft-zimmermann-tcpm-reordering-detection-02 (work in progress), November 2014.

[RFC7765] Hurtig, P., Brunstrom, A., Petlund, A., and M. Welzl, "TCP and SCTP RTO Restart", February 2016.

[SCWA99] Savage, S., Cardwell, N., Wetherall, D., and T. Anderson, "TCP Congestion Control With a Misbehaving Receiver", ACM Computer Communication Review, 29(5) , 1999.

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

[TLP] Dukkipati, N., Cardwell, N., Cheng, Y., and M. Mathis, "Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Drops", draft-dukkipati-tcpm-tcp-loss-probe-01 (work in progress), August 2013.

Authors’ Addresses

Yuchung Cheng
Google, Inc
1600 Amphitheater Parkway
Mountain View, California 94043
USA

Email: ycheng@google.com

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

Email: ncardwell@google.com

Nandita Dukkipati
Google, Inc
1600 Amphitheater Parkway
Mountain View, California 94043

Email: nanditad@google.com

Priyaranjan Jha
Google, Inc
1600 Amphitheater Parkway
Mountain View, California 94043

Email: priyarjha@google.com

TCP Maintenance Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 6, 2021

Y. Cheng
N. Cardwell
N. Dukkipati
P. Jha
Google, Inc

November 2, 2020

The RACK-TLP loss detection algorithm for TCP
draft-ietf-tcpm-rack-13

Abstract

This document presents the RACK-TLP loss detection algorithm for TCP. RACK-TLP uses per-segment transmit timestamps and selective acknowledgements (SACK) and has two parts: RACK ("Recent ACKnowledgment") starts fast recovery quickly using time-based inferences derived from ACK feedback. TLP ("Tail Loss Probe") leverages RACK and sends a probe packet to trigger ACK feedback to avoid retransmission timeout (RTO) events. Compared to the widely used DUPACK threshold approach, RACK-TLP detects losses more efficiently when there are application-limited flights of data, lost retransmissions, or data packet reordering events. It is intended to be an alternative to the DUPACK threshold approach.

Status of This Memo

Copyright Notice

Table of Contents

1. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here. 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. Introduction

This document presents RACK-TLP, a TCP loss detection algorithm that improves upon the widely implemented DUPACK counting approach in [RFC5681][RFC6675], and that is RECOMMENDED to be used as an alternative to that earlier approach. RACK-TLP has two parts: RACK ("Recent ACKnowledgment") detects losses quickly using time-based inferences derived from ACK feedback. TLP ("Tail Loss Probe") triggers ACK feedback by quickly sending a probe segment, to avoid retransmission timeout (RTO) events.

2.1. Background

In traditional TCP loss recovery algorithms [RFC5681][RFC6675], a sender starts fast recovery when the number of DUPACKs received reaches a threshold (DupThresh) that defaults to 3 (this approach is referred to as DUPACK-counting in the rest of the document). The sender also halves the congestion window during the recovery. The rationale behind the partial window reduction is that congestion does not seem severe since ACK clocking is still maintained. The time elapsed in fast recovery can be just one round-trip, e.g. if the sender uses SACK-based recovery [RFC6675] and the number of lost segments is small.

If fast recovery is not triggered, or triggers but fails to repair all the losses, then the sender resorts to RTO recovery. The RTO timer interval is conservatively the smoothed RTT (SRTT) plus four times the RTT variation, and is lower bounded to 1 second [RFC6298].

Upon RTO timer expiration, the sender retransmits the first unacknowledged segment and resets the congestion window to the LOSS WINDOW value (by default 1 full-size segment [RFC5681]). The rationale behind the congestion window reset is that an entire flight of data was lost, and the ACK clock was lost, so this deserves a cautious response. The sender then retransmits the rest of the data following the slow start algorithm [RFC5681]. The time elapsed in RTO recovery is one RTO interval plus the number of round-trips needed to repair all the losses.

2.2. Motivation

Fast Recovery is the preferred form of loss recovery because it can potentially recover all losses in the time scale of a single round trip, with only a fractional congestion window reduction. RTO recovery and congestion window reset should ideally be the last resort, only used when the entire flight is lost. However, in addition to losing an entire flight of data, the following situations can unnecessarily resort to RTO recovery with traditional TCP loss recovery algorithms [RFC5681][RFC6675]:

1. Packet drops for short flows or at the end of an application data flight. When the sender is limited by the application (e.g. structured request/response traffic), segments lost at the end of the application data transfer often can only be recovered by RTO. Consider an example of losing only the last segment in a flight of 100 segments. Lacking any DUPACK, the sender RTO expires and reduces the congestion window to 1, and raises the congestion window to just 2 after the loss repair is acknowledged. In contrast, any single segment loss occurring between the first and the 97th segment would result in fast recovery, which would only cut the window in half.

2. Lost retransmissions. Heavy congestion or traffic policers can cause retransmissions to be lost. Lost retransmissions cause a resort to RTO recovery, since DUPACK-counting does not detect the loss of the retransmissions. Then the slow start after RTO recovery could cause burst losses again that severely degrades performance [POLICER16].

3. Packet reordering. Link-layer protocols (e.g., 802.11 block ACK), link bonding, or routers' internal load-balancing (e.g., ECMP) can deliver TCP segments out of order. The degree of such reordering is usually within the order of the path round trip time. If the reordering degree is beyond DupThresh, the DUPACK-counting can cause a spurious fast recovery and unnecessary congestion window reduction. To mitigate the issue, [RFC4653] adjusts DupThresh to half of the inflight size to tolerate the

higher degree of reordering. However if more than half of the inflight is lost, then the sender has to resort to RTO recovery.

3. RACK-TLP high-level design

RACK-TLP allows senders to recover losses more effectively in all three scenarios described in the previous section. There are two design principles behind RACK-TLP. The first principle is to detect losses via ACK events as much as possible, to repair losses at round-trip time-scales. The second principle is to gently probe the network to solicit additional ACK feedback, to avoid RTO expiration and subsequent congestion window reset. At a high level, the two principles are implemented in RACK and TLP, respectively.

3.1. RACK: time-based loss inferences from ACKs

The rationale behind RACK is that if a segment is delivered out of order, then the segments sent chronologically before that were either lost or reordered. This concept is not fundamentally different from [RFC5681][RFC6675][FACK]. RACK's key innovation is using per-segment transmission timestamps and widely-deployed SACK [RFC2018] options to conduct time-based inferences, instead of inferring losses by counting ACKs or SACKed sequences. Time-based inferences are more robust than DUPACK-counting approaches because they have no dependence on flight size, and thus are effective for application-limited traffic.

Conceptually, RACK puts a virtual timer for every data segment sent (including retransmissions). Each timer expires dynamically based on the latest RTT measurements plus an additional delay budget to accommodate potential packet reordering (called the reordering window). When a segment's timer expires, RACK marks the corresponding segment lost for retransmission.

In reality, as an algorithm, RACK does not arm a timer for every segment sent because it's not necessary. Instead the sender records the most recent transmission time of every data segment sent, including retransmissions. For each ACK received, the sender calculates the latest RTT measurement (if eligible) and adjusts the expiration time of every segment sent but not yet delivered. If a segment has expired, RACK marks it lost.

Since the time-based logic of RACK applies equally to retransmissions and original transmissions, it can detect lost retransmissions as well. If a segment has been retransmitted but its most recent (re)transmission timestamp has expired, then after a reordering window it's marked lost.

3.2. TLP: sending one segment to probe losses quickly with RACK

RACK infers losses from ACK feedback; however, in some cases ACKs are sparse, particularly when the inflight is small or when the losses are high. In some challenging cases the last few segments in a flight are lost. With [RFC5681] or [RFC6675] the sender's RTO would expire and reset the congestion window, when in reality most of the flight has been delivered.

Consider an example where a sender with a large congestion window transmits 100 new data segments after an application write, and only the last three segments are lost. Without RACK-TLP, the RTO expires, the sender retransmits the first unacknowledged segment, and the congestion window slow-starts from 1. After all the retransmits are acknowledged the congestion window has been increased to 4. The total delivery time for this application transfer is three RTTs plus one RTO, a steep cost given that only a tiny fraction of the flight was lost. If instead the losses had occurred three segments sooner in the flight, then fast recovery would have recovered all losses within one round-trip and would have avoided resetting the congestion window.

Fast Recovery would be preferable in such scenarios; TLP is designed to trigger the feedback RACK needed to enable that. After the last (100th) segment was originally sent, TLP sends the next available (new) segment or retransmits the last (highest-sequenced) segment in two round-trips to probe the network, hence the name "Tail Loss Probe". The successful delivery of the probe would solicit an ACK. RACK uses this ACK to detect that the 98th and 99th segments were lost, trigger fast recovery, and retransmit both successfully. The total recovery time is four RTTs, and the congestion window is only partially reduced instead of being fully reset. If the probe was also lost then the sender would invoke RTO recovery resetting the congestion window.

3.3. RACK-TLP: reordering resilience with a time threshold

3.3.1. Reordering design rationale

Upon receiving an ACK indicating an out-of-order data delivery, a sender cannot tell immediately whether that out-of-order delivery was a result of reordering or loss. It can only distinguish between the two in hindsight if the missing sequence ranges are filled in later without retransmission. Thus a loss detection algorithm needs to budget some wait time -- a reordering window -- to try to disambiguate packet reordering from packet loss.

The reordering window in the DUPACK-counting approach is implicitly defined as the elapsed time to receive acknowledgements for DupThresh-worth of out-of-order deliveries. This approach is effective if the network reordering degree (in sequence distance) is smaller than DupThresh and at least DupThresh segments after the loss are acknowledged. For cases where the reordering degree is larger than the default DupThresh of 3 packets, one alternative is to dynamically adapt DupThresh based on the FlightSize (e.g., the sender adjusts the DUPTRESH to half of the FlightSize). However, this does not work well with the following two types of reordering:

1. Application-limited flights where the last non-full-sized segment is delivered first and then the remaining full-sized segments in the flight are delivered in order. This reordering pattern can occur when segments traverse parallel forwarding paths. In such scenarios the degree of reordering in packet distance is one segment less than the flight size.

2. A flight of segments that are delivered partially out of order. One cause for this pattern is wireless link-layer retransmissions with an inadequate reordering buffer at the receiver. In such scenarios, the wireless sender sends the data packets in order initially, but some are lost and then recovered by link-layer retransmissions; the wireless receiver delivers the TCP data packets in the order they are received, due to the inadequate reordering buffer. The random wireless transmission errors in such scenarios cause the reordering degree, expressed in packet distance, to have highly variable values up to the flight size.

In the above two cases the degree of reordering in packet distance is highly variable, making DUPACK-counting approach ineffective including dynamic adaptation variants like [RFC4653]. Instead the degree of reordering in time difference in such cases is usually within a single round-trip time. This is because the packets either traverse slightly disjoint paths with similar propagation delays or are repaired quickly by the local access technology. Hence, using a time threshold instead of packet threshold strikes a middle ground, allowing a bounded degree of reordering resilience while still allowing fast recovery. This is the rationale behind the RACK-TLP reordering resilience design.

Specifically, RACK-TLP introduces a new dynamic reordering window parameter in time units, and the sender considers a data segment S lost if both conditions are met:

1. Another data segment sent later than S has been delivered

2. S has not been delivered after the estimated round-trip time plus the reordering window

Note that condition (1) implies at least one round-trip of time has elapsed since S has been sent.

3.3.2. Reordering window adaptation

The RACK reordering window adapts to the measured duration of reordering events, within reasonable and specific bounds to disincentivize excessive reordering. More specifically, the sender sets the reordering window as follows:

1. The reordering window SHOULD be set to zero if no reordering has been observed on the connection so far, and either (a) three segments have been delivered out of order since the last recovery or (b) the sender is already in fast or RTO recovery. Otherwise, the reordering window SHOULD start from a small fraction of the round trip time, or zero if no round trip time estimate is available.

2. The RACK reordering window SHOULD adaptively increase (using the algorithm in "Step 4: Update RACK reordering window", below) if the sender receives a Duplicate Selective Acknowledgement (DSACK) option [RFC2883]. Receiving a DSACK suggests the sender made a spurious retransmission, which may have been due to the reordering window being too small.

3. The RACK reordering window MUST be bounded and this bound SHOULD be SRTT.

Rules 2 and 3 are required to adapt to reordering caused by dynamics such as the prolonged link-layer loss recovery episodes described earlier. Each increase in the reordering window requires a new round trip where the sender receives a DSACK; thus, depending on the extent of reordering, it may take multiple round trips to fully adapt.

For short flows, the low initial reordering window helps recover losses quickly, at the risk of spurious retransmissions. The rationale is that spurious retransmissions for short flows are not expected to produce excessive additional network traffic. For long flows the design tolerates reordering within a round trip. This handles reordering in small time scales (reordering within the round-trip time of the shortest path).

However, the fact that the initial reordering window is low, and the reordering window's adaptive growth is bounded, means that there will continue to be a cost to reordering that disincentivizes excessive reordering.

3.4. An Example of RACK-TLP in Action: fast recovery

The following example in figure 1 illustrates the RACK-TLP algorithm in action:

```
Event  TCP DATA SENDER                               TCP DATA RECEIVER
_____  ____________________________________________________________
  1.   Send P0, P1, P2, P3          -->
       [P1, P2, P3 dropped by network]

  2.                                <--         Receive P0, ACK P0

  3a.  2RTTs after (2), TLP timer fires
  3b.  TLP: retransmits P3          -->

  4.                                <--        Receive P3, SACK P3

  5a.  Receive SACK for P3
  5b.  RACK: marks P1, P2 lost
  5c.  Retransmit P1, P2            -->
       [P1 retransmission dropped by network]

  6.                                <--   Receive P2, SACK P2 & P3

  7a.  RACK: marks P1 retransmission lost
  7b.  Retransmit P1                -->

  8.                                <--         Receive P1, ACK P3
```

Figure 1. RACK-TLP protocol example

Figure 1, above, illustrates a sender sending four segments (P1, P2, P3, P4) and losing the last three segments. After two round-trips, TLP sends a loss probe, retransmitting the last segment, P3, to solicit SACK feedback and restore the ACK clock (event 3). The delivery of P3 enables RACK to infer (event 5b) that P1 and P2 were likely lost, because they were sent before P3. The sender then retransmits P1 and P2. Unfortunately, the retransmission of P1 is lost again. However, the delivery of the retransmission of P2 allows RACK to infer that the retransmission of P1 was likely lost (event 7a), and hence P1 should be retransmitted (event 7b).

3.5. An Example of RACK-TLP in Action: RTO

In addition to enhancing fast recovery, RACK improves the accuracy of RTO recovery by reducing spurious retransmissions.

Without RACK, upon RTO timer expiration the sender marks all the unacknowledged segments lost. This approach can lead to spurious retransmissions. For example, consider a simple case where one segment was sent with an RTO of 1 second, and then the application writes more data, causing a second and third segment to be sent right before the RTO of the first segment expires. Suppose only the first segment is lost. Without RACK, upon RTO expiration the sender marks all three segments as lost and retransmits the first segment. When the sender receives the ACK that selectively acknowledges the second segment, the sender spuriously retransmits the third segment.

With RACK, upon RTO timer expiration the only segment automatically marked lost is the first segment (since it was sent an RTO ago); for all the other segments RACK only marks the segment lost if at least one round trip has elapsed since the segment was transmitted. Consider the previous example scenario, this time with RACK. With RACK, when the RTO expires the sender only marks the first segment as lost, and retransmits that segment. The other two very recently sent segments are not marked lost, because they were sent less than one round trip ago and there were no ACKs providing evidence that they were lost. When the sender receives the ACK that selectively acknowledges the second segment, the sender would not retransmit the third segment but rather would send any new segments (if allowed by congestion window and receive window).

In the above example, if the sender were to send a large burst of segments instead of two segments right before RTO, without RACK the sender may spuriously retransmit almost the entire flight. Note that the Eifel protocol [RFC3522] cannot prevent this issue because it can only detect spurious RTO episodes. In this example the RTO itself was not spurious.

3.6. Design Summary

To summarize, RACK-TLP aims to adapt to small time-varying degrees of reordering, quickly recover most losses within one to two round trips, and avoid costly RTO recoveries. In the presence of reordering, the adaptation algorithm can impose sometimes-needless delays when it waits to disambiguate loss from reordering, but the penalty for waiting is bounded to one round trip and such delays are confined to flows long enough to have observed reordering.

4. Requirements

The reader is expected to be familiar with the definitions given in the TCP congestion control [RFC5681] and selective acknowledgment [RFC2018][RFC6675] RFCs. RACK-TLP has the following requirements:

1. The connection MUST use selective acknowledgment (SACK) options [RFC2018], and the sender MUST keep SACK scoreboard information on a per-connection basis ("SACK scoreboard" has the same meaning here as in [RFC6675] section 3).

2. For each data segment sent, the sender MUST store its most recent transmission time with a timestamp whose granularity that is finer than 1/4 of the minimum RTT of the connection. At the time of writing, microsecond resolution is suitable for intra-datacenter traffic and millisecond granularity or finer is suitable for the Internet. Note that RACK-TLP can be implemented with TSO (TCP Segmentation Offload) support by having multiple segments in a TSO aggregate share the same timestamp.

3. RACK DSACK-based reordering window adaptation is RECOMMENDED but is not required.

4. TLP requires RACK.

5. Definitions

The reader is expected to be familiar with the variables of SND.UNA, SND.NXT, SEG.ACK, and SEG.SEQ in [RFC793], SMSS, FlightSize in [RFC5681], DupThresh in [RFC6675], RTO and SRTT in [RFC6298]. A RACK-TLP implementation needs to store new per-segment and per-connection state, described below.

5.1. Per-segment variables

Theses variables indicate the status of the most recent transmission of a data segment:

"Segment.lost" is true if the most recent (re)transmission of the segment has been marked lost and needs to be retransmitted. False otherwise.

"Segment.retransmitted" is true if the segment has ever been retransmitted. False otherwise.

"Segment.xmit_ts" is the time of the last transmission of a data segment, including retransmissions, if any, with a clock granularity specified in the Requirements section. A maximum value INFINITE_TS indicates an invalid timestamp that represents that the Segment is not currently in flight.

"Segment.end_seq" is the next sequence number after the last sequence number of the data segment.

5.2. Per-connection variables

"RACK.segment". Among all the segments that have been either selectively or cumulatively acknowledged, RACK.segment is the one that was sent most recently (including retransmissions).

"RACK.xmit_ts" is the latest transmission timestamp of RACK.segment.

"RACK.end_seq" is the Segment.end_seq of RACK.segment.

"RACK.ack_ts" is the time when the full sequence range of RACK.segment was selectively or cumulatively acknowledged.

"RACK.segs_sacked" returns the total number of segments selectively acknowledged in the SACK scoreboard.

"RACK.fack" is the highest selectively or cumulatively acknowledged sequence (i.e. forward acknowledgement).

"RACK.min_RTT" is the estimated minimum round-trip time (RTT) of the connection.

"RACK.rtt" is the RTT of the most recently delivered segment on the connection (either cumulatively acknowledged or selectively acknowledged) that was not marked invalid as a possible spurious retransmission.

"RACK.reordering_seen" indicates whether the sender has detected data segment reordering event(s).

"RACK.reo_wnd" is a reordering window computed in the unit of time used for recording segment transmission times. It is used to defer the moment at which RACK marks a segment lost.

"RACK.dsack" indicates if a DSACK option has been received since the last RACK.reo_wnd change.

"RACK.reo_wnd_mult" is the multiplier applied to adjust RACK.reo_wnd.

"RACK.reo_wnd_persist" is the number of loss recoveries before resetting RACK.reo_wnd.

"RACK.rtt_seq" is the SND.NXT when RACK.rtt is updated.

"TLP.is_retrans": a boolean indicating whether there is an unacknowledged TLP retransmission.

"TLP.end_seq": the value of SND.NXT at the time of sending a TLP retransmission.

"TLP.max_ack_delay": sender’s maximum delayed ACK timer budget.

Per-connection timers

"RACK reordering timer": a timer that allows RACK to wait for reordering to resolve, to try to disambiguate reordering from loss, when some out-of-order segments are marked as SACKed.

"TLP PTO": a timer event indicating that an ACK is overdue and the sender should transmit a TLP segment, to solicit SACK or ACK feedback.

These timers augment the existing timers maintained by a sender, including the RTO timer [RFC6298]. A RACK-TLP sender arms one of these three timers -- RACK reordering timer, TLP PTO timer, or RTO timer -- when it has unacknowledged segments in flight. The implementation can simplify managing all three timers by multiplexing a single timer among them with an additional variable to indicate the event to invoke upon the next timer expiration.

6. RACK Algorithm Details

6.1. Upon transmitting a data segment

Upon transmitting a new segment or retransmitting an old segment, record the time in Segment.xmit_ts and set Segment.lost to FALSE. Upon retransmitting a segment, set Segment.retransmitted to TRUE.

```
RACK_transmit_new_data(Segment):
        Segment.xmit_ts = Now()
        Segment.lost = FALSE

RACK_retransmit_data(Segment):
        Segment.retransmitted = TRUE
        Segment.xmit_ts = Now()
        Segment.lost = FALSE
```

6.2. Upon receiving an ACK

Step 1: Update RACK.min_RTT.

Use the RTT measurements obtained via [RFC6298] or [RFC7323] to update the estimated minimum RTT in RACK.min_RTT. The sender SHOULD track a windowed min-filtered estimate of recent RTT measurements that can adapt when migrating to significantly longer paths, rather than a simple global minimum of all RTT measurements.

Step 2: Update state for most recently sent segment that has been delivered

In this step, RACK updates the states that track the most recently sent segment that has been delivered: RACK.segment; RACK maintains its latest transmission timestamp in RACK.xmit_ts and its highest sequence number in RACK.end_seq. These two variables are used, in later steps, to estimate if some segments not yet delivered were likely lost. Given the information provided in an ACK, each segment cumulatively ACKed or SACKed is marked as delivered in the scoreboard. Since an ACK can also acknowledge retransmitted data segments, and retransmissions can be spurious, the sender needs to take care to avoid spurious inferences. For example, if the sender were to use timing information from a spurious retransmission, the RACK.rtt could be vastly underestimated.

To avoid spurious inferences, ignore a segment as invalid if any of its sequence range has been retransmitted before and either of two conditions is true:

1. The Timestamp Echo Reply field (TSecr) of the ACK's timestamp option [RFC7323], if available, indicates the ACK was not acknowledging the last retransmission of the segment.

2. The segment was last retransmitted less than RACK.min_rtt ago.

The second check is a heuristic when the TCP Timestamp option is not available, or when the round trip time is less than the TCP Timestamp clock granularity.

Among all the segments newly ACKed or SACKed by this ACK that pass the checks above, update the RACK.rtt to be the RTT sample calculated using this ACK. Furthermore, record the most recent Segment.xmit_ts in RACK.xmit_ts if it is ahead of RACK.xmit_ts. If Segment.xmit_ts equals RACK.xmit_ts (e.g. due to clock granularity limits) then compare Segment.end_seq and RACK.end_seq to break the tie.

Step 2 may be summarized in pseudocode as:

```
RACK_sent_after(t1, seq1, t2, seq2):
    If t1 > t2:
        Return true
    Else if t1 == t2 AND seq1 > seq2:
        Return true
    Else:
        Return false

RACK_update():
    For each Segment newly acknowledged cumulatively or selectively:
        rtt = Now() - Segment.xmit_ts
        If Segment.retransmitted is TRUE:
            If ACK.ts_option.echo_reply < Segment.xmit_ts:
               Return
            If rtt < RACK.min_rtt:
               Return

        RACK.rtt = rtt
        If RACK_sent_after(Segment.xmit_ts, Segment.end_seq
                           RACK.xmit_ts, RACK.end_seq):
            RACK.xmit_ts = Segment.xmit_ts
```

Step 3: Detect data segment reordering

To detect reordering, the sender looks for original data segments being delivered out of order. To detect such cases, the sender tracks the highest sequence selectively or cumulatively acknowledged in the RACK.fack variable. The name "fack" stands for the most "Forward ACK" (this term is adopted from [FACK]). If a never-retransmitted segment that’s below RACK.fack is (selectively or cumulatively) acknowledged, it has been delivered out of order. The sender sets RACK.reordering_seen to TRUE if such segment is identified.

```
RACK_detect_reordering():
    For each Segment newly acknowledged cumulatively or selectively:
        If Segment.end_seq > RACK.fack:
            RACK.fack = Segment.end_seq
        Else if Segment.end_seq < RACK.fack AND
                Segment.retransmitted is FALSE:
            RACK.reordering_seen = TRUE
```

Step 4: Update RACK reordering window

The RACK reordering window, RACK.reo_wnd, serves as an adaptive allowance for settling time before marking a segment lost. This step documents a detailed algorithm that follows the principles outlined in the ‘‘Reordering window adaptation’’ section.

If no reordering has been observed, based on the previous step, then one way the sender can enter Fast Recovery is when the number of SACKed segments matches or exceeds DupThresh (similar to RFC6675). Furthermore, when no reordering has been observed the RACK.reo_wnd is set to 0 both upon entering and during Fast Recovery or RTO recovery.

Otherwise, if some reordering has been observed, then RACK does not trigger Fast Recovery based on DupThresh.

Whether or not reordering has been observed, RACK uses the reordering window to assess whether any segments can be marked lost. As a consequence, the sender also enters Fast Recovery when there are any number of SACKed segments as long as the reorder window has passed for some non-SACKed segments.

When the reordering window is not set to 0, it starts with a conservative RACK.reo_wnd of RACK.min_RTT/4. This value was chosen because Linux TCP used the same factor in its implementation to delay Early Retransmit [RFC5827] to reduce spurious loss detections in the presence of reordering, and experience showed this worked reasonably well [DMCG11].

However, the reordering detection in the previous step, Step 3, has a self-reinforcing drawback when the reordering window is too small to cope with the actual reordering. When that happens, RACK could spuriously mark reordered segments lost, causing them to be retransmitted. In turn, the retransmissions can prevent the necessary conditions for Step 3 to detect reordering, since this mechanism requires ACKs or SACKs for only segments that have never been retransmitted. In some cases such scenarios can persist, causing RACK to continue to spuriously mark segments lost without realizing the reordering window is too small.

To avoid the issue above, RACK dynamically adapts to higher degrees of reordering using DSACK options from the receiver. Receiving an ACK with a DSACK option indicates a possible spurious retransmission, suggesting that RACK.reo_wnd may be too small. The RACK.reo_wnd increases linearly for every round trip in which the sender receives some DSACK option, so that after N distinct round trips in which a DSACK is received, the RACK.reo_wnd becomes (N+1) * min_RTT / 4, with an upper-bound of SRTT.

If the reordering is temporary then a large adapted reordering window would unnecessarily delay loss recovery later. Therefore, RACK persists using the inflated RACK.reo_wnd for up to 16 loss recoveries, after which it resets RACK.reo_wnd to its starting value, min_RTT / 4. The downside of resetting the reordering window is the risk of triggering spurious fast recovery episodes if the reordering remains high. The rationale for this approach is to bound such spurious recoveries to approximately once every 16 recoveries (less than 7%).

To track the linear scaling factor for the adaptive reordering window, RACK uses the variable RACK.reo_wnd_mult, which is initialized to 1 and adapts with observed reordering.

The following pseudocode implements the above algorithm for updating the RACK reordering window:

```
RACK_update_reo_wnd():

    /* DSACK-based reordering window adaptation */
    If RACK.dsack_round is not None AND
       SND.UNA >= RACK.dsack_round:
        RACK.dsack_round = None
    /* Grow the reordering window per round that sees DSACK.
       Reset the window after 16 DSACK-free recoveries */
    If RACK.dsack_round is None AND
       any DSACK option is present on latest received ACK:
        RACK.dsack_round = SND.NXT
        RACK.reo_wnd_mult += 1
        RACK.reo_wnd_persist = 16
    Else if exiting Fast or RTO recovery:
        RACK.reo_wnd_persist -= 1
        If RACK.reo_wnd_persist <= 0:
            RACK.reo_wnd_mult = 1

    If RACK.reordering_seen is FALSE:
        If in Fast or RTO recovery:
            Return 0
        Else if RACK.segs_sacked >= DupThresh:
            Return 0
    Return min(RACK.min_RTT / 4 * RACK.reo_wnd_mult, SRTT)
```

Step 5: Detect losses.

For each segment that has not been SACKed, RACK considers that segment lost if another segment that was sent later has been delivered, and the reordering window has passed. RACK considers the reordering window to have passed if the RACK.segment was sent sufficiently after the segment in question, or a sufficient time has elapsed since the RACK.segment was S/ACKed, or some combination of the two. More precisely, RACK marks a segment lost if:

```
 RACK.xmit_ts >= Segment.xmit_ts
        AND
 RACK.xmit_ts - Segment.xmit_ts + (now - RACK.ack_ts) >= RACK.reo_wnd
```

Solving this second condition for "now", the moment at which a segment is marked lost, yields:

```
now >= Segment.xmit_ts + RACK.reo_wnd + (RACK.ack_ts - RACK.xmit_ts)
```

Then (RACK.ack_ts - RACK.xmit_ts) is the round trip time of the most recently (re)transmitted segment that's been delivered. When segments are delivered in order, the most recently (re)transmitted segment that's been delivered is also the most recently delivered, hence RACK.rtt == RACK.ack_ts - RACK.xmit_ts. But if segments were reordered, then the segment delivered most recently was sent before the most recently (re)transmitted segment. Hence RACK.rtt > (RACK.ack_ts - RACK.xmit_ts).

Since RACK.RTT >= (RACK.ack_ts - RACK.xmit_ts), the previous equation reduces to saying that the sender can declare a segment lost when:

```
now >= Segment.xmit_ts + RACK.reo_wnd + RACK.rtt
```

In turn, that is equivalent to stating that a RACK sender should declare a segment lost when:

```
Segment.xmit_ts + RACK.rtt + RACK.reo_wnd - now <= 0
```

Note that if the value on the left hand side is positive, it represents the remaining wait time before the segment is deemed lost. But this risks a timeout (RTO) if no more ACKs come back (e.g., due to losses or application-limited transmissions) to trigger the marking. For timely loss detection, the sender is RECOMMENDED to install a reordering timer. This timer expires at the earliest moment when RACK would conclude that all the unacknowledged segments within the reordering window were lost.

The following pseudocode implements the algorithm above. When an ACK is received or the RACK reordering timer expires, call RACK_detect_loss_and_arm_timer(). The algorithm breaks timestamp ties by using the TCP sequence space, since high-speed networks often have multiple segments with identical timestamps.

```
RACK_detect_loss():
    timeout = 0
    RACK.reo_wnd = RACK_update_reo_wnd()
    For each segment, Segment, not acknowledged yet:
        If RACK_sent_after(RACK.xmit_ts, RACK.end_seq,
                           Segment.xmit_ts, Segment.end_seq):
            remaining = Segment.xmit_ts + RACK.rtt +
                        RACK.reo_wnd - Now()
            If remaining <= 0:
                Segment.lost = TRUE
                Segment.xmit_ts = INFINITE_TS
            Else:
                timeout = max(remaining, timeout)
    Return timeout

RACK_detect_loss_and_arm_timer():
    timeout = RACK_detect_loss()
    If timeout != 0
        Arm the RACK timer to call
        RACK_detect_loss_and_arm_timer() after timeout
```

As an optimization, an implementation can choose to check only segments that have been sent before RACK.xmit_ts. This can be more efficient than scanning the entire SACK scoreboard, especially when there are many segments in flight. The implementation can use a separate doubly-linked list ordered by Segment.xmit_ts and inserts a segment at the tail of the list when it is (re)transmitted, and removes a segment from the list when it is delivered or marked lost. In Linux TCP this optimization improved CPU usage by orders of magnitude during some fast recovery episodes on high-speed WAN networks.

6.3. Upon RTO expiration

Upon RTO timer expiration, RACK marks the first outstanding segment as lost (since it was sent an RTO ago); for all the other segments RACK only marks the segment lost if the time elapsed since the segment was transmitted is at least the sum of the recent RTT and the reordering window.

```
RACK_mark_losses_on_RTO():
    For each segment, Segment, not acknowledged yet:
        If SEG.SEQ == SND.UNA OR
           Segment.xmit_ts + RACK.rtt + RACK.reo_wnd - Now() <= 0:
            Segment.lost = TRUE
```

7. TLP Algorithm Details

7.1. Initializing state

Reset TLP.is_retrans and TLP.end_seq when initiating a connection, fast recovery, or RTO recovery.

```
TLP_init():
    TLP.end_seq = None
    TLP.is_retrans = false
```

7.2. Scheduling a loss probe

The sender schedules a loss probe timeout (PTO) to transmit a segment during the normal transmission process. The sender SHOULD start or restart a loss probe PTO timer after transmitting new data (that was not itself a loss probe) or upon receiving an ACK that cumulatively acknowledges new data, unless it is already in fast recovery, RTO recovery, or the sender has segments delivered out-of-order (i.e. RACK.segs_sacked is not zero). These conditions are excluded because they are addressed by similar mechanisms, like Limited Transmit [RFC3042], the RACK reordering timer, and F-RTO [RFC5682].

The sender calculates the PTO interval by taking into account a number of factors.

First, the default PTO interval is 2*SRTT. By that time, it is prudent to declare that an ACK is overdue, since under normal circumstances, i.e. no losses, an ACK typically arrives in one SRTT. Choosing PTO to be exactly an SRTT would risk causing spurious probes, given that network and end-host delay variance can cause an ACK to be delayed beyond SRTT. Hence the PTO is conservatively chosen to be the next integral multiple of SRTT.

Second, when there is no SRTT estimate available, the PTO SHOULD be 1 second. This conservative value corresponds to the RTO value when no SRTT is available, per [RFC6298].

Third, when FlightSize is one segment, the sender MAY inflate PTO by TLP.max_ack_delay to accommodate a potential delayed acknowledgment and reduce the risk of spurious retransmissions. The actual value of TLP.max_ack_delay is implementation-specific.

Finally, if the time at which an RTO would fire (here denoted "TCP_RTO_expiration()") is sooner than the computed time for the PTO, then the sender schedules a TLP to be sent at that RTO time.

Summarizing these considerations in pseudocode form, a sender SHOULD use the following logic to select the duration of a PTO:

```
TLP_calc_PTO():
    If SRTT is available:
        PTO = 2 * SRTT
        If FlightSize is one segment:
           PTO += TLP.max_ack_delay
    Else:
        PTO = 1 sec

    If Now() + PTO > TCP_RTO_expiration():
        PTO = TCP_RTO_expiration() - Now()
```

7.3. Sending a loss probe upon PTO expiration

When the PTO timer expires, the sender SHOULD transmit a previously unsent data segment, if the receive window allows, and increment the FlightSize accordingly. Note that FlightSize could be one packet greater than the congestion window temporarily until the next ACK arrives.

If such a segment is not available, then the sender SHOULD retransmit the highest-sequence segment sent so far and set TLP.is_retrans to true. This segment is chosen to deal with the retransmission ambiguity problem in TCP. Suppose a sender sends N segments, and then retransmits the last segment (segment N) as a loss probe, and then the sender receives a SACK for segment N. As long as the sender waits for the RACK reordering window to expire, it doesn’t matter if that SACK was for the original transmission of segment N or the TLP retransmission; in either case the arrival of the SACK for segment N provides evidence that the N-1 segments preceding segment N were likely lost.

In the case where there is only one original outstanding segment of data (N=1), the same logic (trivially) applies: an ACK for a single outstanding segment tells the sender the N-1=0 segments preceding that segment were lost. Furthermore, whether there are N>1 or N=1 outstanding segments, there is a question about whether the original last segment or its TLP retransmission were lost; the sender estimates whether there was such a loss using TLP recovery detection (see below).

The sender MUST follow the RACK transmission procedures in the ’’Upon Transmitting a Data Segment’’ section (see above) upon sending either a retransmission or new data loss probe. This is critical for detecting losses using the ACK for the loss probe. Furthermore, prior to sending a loss probe, the sender MUST check that there is no other previous loss probe still in flight. This ensures that at any given time the sender has at most one additional packet in flight beyond the congestion window limit. This invariant is maintained using the state variable TLP.end_seq, which indicates the latest unacknowledged TLP loss probe's ending sequence. It is reset when the loss probe has been acknowledged or is deemed lost or irrelevant. After attempting to send a loss probe, regardless of whether a loss probe was sent, the sender MUST re-arm the RTO timer, not the PTO timer, if FlightSize is not zero. This ensures RTO recovery remains the last resort if TLP fails. The following pseudo code summarizes the operations.

```
TLP_send_probe():

    If TLP.end_seq is None:
        TLP.is_retrans = false
        Segment = send buffer segment starting at SND.NXT
        If Segment exists and fits the peer receive window limit:
           /* Transmit the lowest-sequence unsent Segment */
           Transmit Segment
           RACK_transmit_data(Segment)
           TLP.end_seq = SND.NXT
           Increase FlightSize by Segment length
        Else:
           /* Retransmit the highest-sequence Segment sent */
           Segment = send buffer segment ending at SND.NXT
           Transmit Segment
           RACK_retransmit_data(Segment)
           TLP.end_seq = SND.NXT
```

7.4. Detecting losses using the ACK of the loss probe

When there is packet loss in a flight ending with a loss probe, the feedback solicited by a loss probe will reveal one of two scenarios, depending on the pattern of losses.

7.4.1. General case: detecting packet losses using RACK

If the loss probe and the ACK that acknowledges the probe are delivered successfully, RACK-TLP uses this ACK -- just as it would with any other ACK -- to detect if any segments sent prior to the probe were dropped. RACK would typically infer that any unacknowledged data segments sent before the loss probe were lost, since they were sent sufficiently far in the past (at least one PTO has elapsed, plus one round-trip for the loss probe to be ACKed). More specifically, RACK_detect_loss() (step 5) would mark those earlier segments as lost. Then the sender would trigger a fast recovery to recover those losses.

7.4.2. Special case: detecting a single loss repaired by the loss probe

If the TLP retransmission repairs all the lost in-flight sequence ranges (i.e. only the last segment in the flight was lost), the ACK for the loss probe appears to be a regular cumulative ACK, which would not normally trigger the congestion control response to this packet loss event. The following TLP recovery detection mechanism examines ACKs to detect this special case to make congestion control respond properly [RFC5681].

After a TLP retransmission, the sender checks for this special case of a single loss that is recovered by the loss probe itself. To accomplish this, the sender checks for a duplicate ACK or DSACK indicating that both the original segment and TLP retransmission arrived at the receiver, meaning there was no loss. If the TLP sender does not receive such an indication, then it MUST assume that either the original data segment, the TLP retransmission, or a corresponding ACK were lost, for congestion control purposes.

If the TLP retransmission is spurious, a receiver that uses DSACK would return an ACK that covers TLP.end_seq with a DSACK option (Case 1). If the receiver does not support DSACK, it would return a DUPACK without any SACK option (Case 2). If the sender receives an ACK matching either case, then the sender estimates that the receiver received both the original data segment and the TLP probe retransmission, and so the sender considers the TLP episode to be done, and records that fact by setting TLP.end_seq to None.

Upon receiving an ACK that covers some sequence number after TLP.end_seq, the sender should have received any ACKs for the original segment and TLP probe retransmission segment. At that time, if the TLP.end_seq is still set, and thus indicates that the TLP probe retransmission remains unacknowledged, then the sender should presume that at least one of its data segments was lost. The sender then SHOULD invoke a congestion control response equivalent to a fast recovery.

More precisely, on each ACK the sender executes the following:

```
TLP_process_ack(ACK):
    If TLP.end_seq is not None AND ACK's ack. number >= TLP.end_seq:
        If not TLP.is_retrans:
            TLP.end_seq = None    /* TLP of new data delivered */
        Else if ACK has a DSACK option matching TLP.end_seq:
            TLP.end_seq = None    /* Case 1, above */
        Else If ACK's ack. number > TLP.end_seq:
            TLP.end_seq = None    /* Repaired the single loss */
            (Invoke congestion control to react to
             the loss event the probe has repaired)
        Else If ACK is a DUPACK without any SACK option:
            TLP.end_seq = None     /* Case 2, above */
```

8. Managing RACK-TLP timers

The RACK reordering, the TLP PTO timer, the RTO and Zero Window Probe (ZWP) timer [RFC793] are mutually exclusive and used in different scenarios. When arming a RACK reordering timer or TLP PTO timer, the sender SHOULD cancel any other pending timer(s). An implementation is to have one timer with an additional state variable indicating the type of the timer.

9. Discussion

9.1. Advantages and disadvantages

The biggest advantage of RACK-TLP is that every data segment, whether it is an original data transmission or a retransmission, can be used to detect losses of the segments sent chronologically prior to it. This enables RACK-TLP to use fast recovery in cases with application-limited flights of data, lost retransmissions, or data segment reordering events. Consider the following examples:

1. Packet drops at the end of an application data flight: Consider a sender that transmits an application-limited flight of three data segments (P1, P2, P3), and P1 and P3 are lost. Suppose the transmission of each segment is at least RACK.reo_wnd after the transmission of the previous segment. RACK will mark P1 as lost when the SACK of P2 is received, and this will trigger the retransmission of P1 as R1. When R1 is cumulatively acknowledged, RACK will mark P3 as lost and the sender will retransmit P3 as R3. This example illustrates how RACK is able to repair certain drops at the tail of a transaction without an RTO recovery. Notice that neither the conventional duplicate ACK threshold [RFC5681], nor [RFC6675], nor the Forward Acknowledgment [FACK] algorithm can detect such losses, because of the required segment or sequence count.

2. Lost retransmission: Consider a flight of three data segments (P1, P2, P3) that are sent; P1 and P2 are dropped. Suppose the transmission of each segment is at least RACK.reo_wnd after the transmission of the previous segment. When P3 is SACKed, RACK will mark P1 and P2 lost and they will be retransmitted as R1 and R2. Suppose R1 is lost again but R2 is SACKed; RACK will mark R1 lost and trigger retransmission again. Again, neither the conventional three duplicate ACK threshold approach, nor [RFC6675], nor the Forward Acknowledgment [FACK] algorithm can detect such losses. And such a lost retransmission can happen when TCP is being rate-limited, particularly by token bucket policers with large bucket depth and low rate limit; in such cases retransmissions are often lost repeatedly because standard congestion control requires multiple round trips to reduce the rate below the policed rate.

3. Packet reordering: Consider a simple reordering event where a flight of segments are sent as (P1, P2, P3). P1 and P2 carry a full payload of MSS octets, but P3 has only a 1-octet payload. Suppose the sender has detected reordering previously and thus RACK.reo_wnd is min_RTT/4. Now P3 is reordered and delivered first, before P1 and P2. As long as P1 and P2 are delivered within min_RTT/4, RACK will not consider P1 and P2 lost. But if P1 and P2 are delivered outside the reordering window, then RACK will still spuriously mark P1 and P2 lost.

The examples above show that RACK-TLP is particularly useful when the sender is limited by the application, which can happen with interactive or request/response traffic. Similarly, RACK still works when the sender is limited by the receive window, which can happen with applications that use the receive window to throttle the sender.

RACK-TLP works more efficiently with TCP Segmentation Offload (TSO) compared to DUPACK-counting. RACK always marks the entire TSO aggregate lost because the segments in the same TSO aggregate have the same transmission timestamp. By contrast, the algorithms based on sequence counting (e.g., [RFC6675][RFC5681]) may mark only a subset of segments in the TSO aggregate lost, forcing the stack to perform expensive fragmentation of the TSO aggregate, or to selectively tag individual segments lost in the scoreboard.

The main drawback of RACK-TLP is the additional states required compared to DUPACK-counting. RACK requires the sender to record the transmission time of each segment sent at a clock granularity that is finer than 1/4 of the minimum RTT of the connection. TCP implementations that record this already for RTT estimation do not require any new per-packet state. But implementations that are not yet recording segment transmission times will need to add per-packet internal state (expected to be either 4 or 8 octets per segment or TSO aggregate) to track transmission times. In contrast, [RFC6675] loss detection approach does not require any per-packet state beyond the SACK scoreboard; this is particularly useful on ultra-low RTT networks where the RTT may be less than the sender TCP clock granularity (e.g. inside data-centers). Another disadvantage is the reordering timer may expire prematurely (like any other retransmission timer) to cause higher spurious retransmission especially if DSACK is not supported.

9.2. Relationships with other loss recovery algorithms

The primary motivation of RACK-TLP is to provide a general alternative to some of the standard loss recovery algorithms [RFC5681][RFC6675][RFC5827][RFC4653]. [RFC5827][RFC4653] dynamically adjusts the duplicate ACK threshold based on the current or previous flight sizes. RACK-TLP takes a different approach by using a time-based reordering window. RACK-TLP can be seen as an extended Early Retransmit [RFC5827] without a FlightSize limit but with an additional reordering window. [FACK] considers an original segment to be lost when its sequence range is sufficiently far below the highest SACKed sequence. In some sense RACK-TLP can be seen as a generalized form of FACK that operates in time space instead of sequence space, enabling it to better handle reordering, application-limited traffic, and lost retransmissions.

RACK-TLP is compatible with the standard RTO [RFC6298], RTO-restart [RFC7765], F-RTO [RFC5682] and Eifel algorithms [RFC3522]. This is because RACK-TLP only detects loss by using ACK events. It neither changes the RTO timer calculation nor detects spurious RTO.

9.3. Interaction with congestion control

RACK-TLP intentionally decouples loss detection from congestion control. RACK-TLP only detects losses; it does not modify the congestion control algorithm [RFC5681][RFC6937]. A segment marked lost by RACK-TLP MUST NOT be retransmitted until congestion control deems this appropriate.

The only exception -- the only way in which RACK-TLP modulates the congestion control algorithm -- is that one outstanding loss probe can be sent even if the congestion window is fully used. However, this temporary over-commit is accounted for and credited in the in-flight data tracked for congestion control, so that congestion control will erase the over-commit upon the next ACK.

If packet losses happen after the reordering window has been increased by DSACK, RACK-TLP may take longer to detect losses than the pure DUPACK-counting approach. In this case TCP may continue to increase the congestion window upon receiving ACKs during this time, making the sender more aggressive.

The following simple example compares how RACK-TLP and non-RACK-TLP loss detection interacts with congestion control: suppose a sender has a congestion window (cwnd) of 20 segments on a SACK-enabled connection. It sends 10 data segments and all of them are lost.

Without RACK-TLP, the sender would time out, reset cwnd to 1, and retransmit the first segment. It would take four round trips (1 + 2 + 4 + 3 = 10) to retransmit all the 10 lost segments using slow start. The recovery latency would be RTO + 4*RTT, with an ending cwnd of 4 segments due to congestion window validation.

With RACK-TLP, a sender would send the TLP after 2*RTT and get a DUPACK, enabling RACK to detect the losses and trigger fast recovery. If the sender implements Proportional Rate Reduction [RFC6937] it would slow start to retransmit the remaining 9 lost segments since the number of segments in flight (0) is lower than the slow start threshold (10). The slow start would again take four round trips (1 + 2 + 4 + 3 = 10) to retransmit all the lost segments. The recovery latency would be 2*RTT + 4*RTT, with an ending cwnd set to the slow start threshold of 10 segments.

The difference in recovery latency (RTO + 4*RTT vs 6*RTT) can be significant if the RTT is much smaller than the minimum RTO (1 second in [RFC6298]) or if the RTT is large. The former case can happen in local area networks, data-center networks, or content distribution networks with deep deployments. The latter case can happen in developing regions with highly congested and/or high-latency networks.

9.4. TLP recovery detection with delayed ACKs

Delayed or stretched ACKs complicate the detection of repairs done by TLP, since with such ACKs the sender takes longer time to receive fewer ACKs 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 above features such a delay, in the form of TLP.max_ack_delay. Furthermore, if the receiver supports DSACK then in the case of a delayed ACK the sender's TLP recovery detection mechanism (see above) can use the DSACK information to infer that the original and TLP retransmission both arrived at the receiver.

If there is ACK loss or a delayed ACK without a DSACK, then this algorithm is conservative, because the sender will reduce the congestion window 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 the congestion window can be prudent.

9.5. RACK for other transport protocols

RACK can be implemented in other transport protocols (e.g., [QUIC-LR]). The [Sprout] loss detection algorithm was also independently designed to use a 10ms reordering window to improve its loss detection.

10. Security Considerations

RACK-TLP algorithm behavior is based on information conveyed in SACK options, so it has security considerations similar to those described in the Security Considerations section of [RFC6675].

Additionally, RACK-TLP has a lower risk profile than [RFC6675] because it is not vulnerable to ACK-splitting attacks [SCWA99]: for an MSS-size segment sent, the receiver or the attacker might send MSS ACKs that SACK or acknowledge one additional byte per ACK. This would not fool RACK. In such a scenario, RACK.xmit_ts would not advance, because all the sequence ranges within the segment were transmitted at the same time, and thus carry the same transmission timestamp. In other words, SACKing only one byte of a segment or SACKing the segment in entirety have the same effect with RACK.

11. IANA Considerations

This document makes no request of IANA.

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

12. Acknowledgments

The authors thank Matt Mathis for his insights in FACK and Michael Welzl for his per-packet timer idea that inspired this work. Eric Dumazet, Randy Stewart, Van Jacobson, Ian Swett, Rick Jones, Jana Iyengar, Hiren Panchasara, Praveen Balasubramanian, Yoshifumi Nishida, Bob Briscoe, Felix Weinrank, Michael Tuexen, Martin Duke, Ilpo Jarvinen, Theresa Enghardt, Mirja Kuehlewind, Gorry Fairhurst, and Yi Huang contributed to the draft or the implementations in Linux, FreeBSD, Windows, and QUIC.

13. References

13.1. Normative References

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

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

[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", January 2001.

[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.

[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.

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

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

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", May 2017.

13.2. Informative References

[DMCG11] Dukkipati, N., Matthis, M., Cheng, Y., and M. Ghobadi, "Proportional Rate Reduction for TCP", ACM SIGCOMM Conference on Internet Measurement , 2011.

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

[POLICER16] Flach, T., Papageorge, P., Terzis, A., Pedrosa, L., Cheng, Y., Karim, T., Katz-Bassett, E., and R. Govindan, "An Analysis of Traffic Policing in the Web", ACM SIGCOMM , 2016.

[QUIC-LR] Iyengar, J. and I. Swett, "QUIC Loss Detection and Congestion Control", draft-ietf-quic-recovery (work in progress), Octobor 2020.

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

[RFC4653] Bhandarkar, S., Reddy, A., Allman, M., and E. Blanton, "Improving the Robustness of TCP to Non-Congestion Events", August 2006.

[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.

[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.

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

[RFC7765] Hurtig, P., Brunstrom, A., Petlund, A., and M. Welzl, "TCP and SCTP RTO Restart", February 2016.

[SCWA99] Savage, S., Cardwell, N., Wetherall, D., and T. Anderson, "TCP Congestion Control With a Misbehaving Receiver", ACM Computer Communication Review, 29(5) , 1999.

[Sprout] Winstein, K., Sivaraman, A., and H. Balakrishnan, "Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks", USENIX Symposium on Networked Systems Design and Implementation (NSDI) , 2013.

Authors' Addresses

Yuchung Cheng
Google, Inc

Email: ycheng@google.com

Neal Cardwell
Google, Inc

Email: ncardwell@google.com

Nandita Dukkipati
Google, Inc

Email: nanditad@google.com

Priyaranjan Jha
Google, Inc

Email: priyarjha@google.com