

LWIG Working Group  
Internet-Draft  
Intended status: Informational  
Expires: December 12, 2018

C. Gomez  
UPC  
J. Crowcroft  
University of Cambridge  
M. Scharf  
Nokia  
June 10, 2018

TCP Usage Guidance in the Internet of Things (IoT)  
draft-ietf-lwig-tcp-constrained-node-networks-03

## Abstract

This document provides guidance on how to implement and use the Transmission Control Protocol (TCP) in Constrained-Node Networks (CNNs), which are a characteristic of the Internet of Things (IoT). Such environments require a lightweight TCP implementation and may not make use of optional functionality. This document explains a number of known and deployed techniques to simplify a TCP stack as well as corresponding tradeoffs. The objective is to help embedded developers with decisions on which TCP features to use.

## Status of This Memo

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

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

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

This Internet-Draft will expire on December 12, 2018.

## Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

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

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

## Table of Contents

1. Introduction . . . . .	3
2. Conventions used in this document . . . . .	4
3. Characteristics of CNNs relevant for TCP . . . . .	4
3.1. Network and link properties . . . . .	4
3.2. Usage scenarios . . . . .	5
3.3. Communication and traffic patterns . . . . .	6
4. TCP implementation and configuration in CNNs . . . . .	6
4.1. Path properties . . . . .	7
4.1.1. Maximum Segment Size (MSS) . . . . .	7
4.1.2. Explicit Congestion Notification (ECN) . . . . .	7
4.1.3. Explicit loss notifications . . . . .	8
4.2. TCP guidance for small windows and buffers . . . . .	8
4.2.1. Single-MSS stacks - benefits and issues . . . . .	8
4.2.2. TCP options for single-MSS stacks . . . . .	9
4.2.3. Delayed Acknowledgments for single-MSS stacks . . . . .	9
4.2.4. RTO estimation for single-MSS stacks . . . . .	10
4.3. General recommendations for TCP in CNNs . . . . .	10
4.3.1. Error recovery and congestion/flow control . . . . .	10
4.3.2. Selective Acknowledgments (SACK) . . . . .	11
4.3.3. Delayed Acknowledgments . . . . .	11
5. TCP usage recommendations in CNNs . . . . .	11
5.1. TCP connection initiation . . . . .	12
5.2. TCP connection lifetime . . . . .	12
5.2.1. Long TCP connection lifetime . . . . .	12
5.2.2. Short TCP connection lifetime . . . . .	12
5.3. Number of parallel connections . . . . .	13
6. Security Considerations . . . . .	13
7. Acknowledgments . . . . .	14
8. Annex. TCP implementations for constrained devices . . . . .	14
8.1. uIP . . . . .	14
8.2. lwIP . . . . .	15
8.3. RIOT . . . . .	15
8.4. OpenWSN . . . . .	16
8.5. TinyOS . . . . .	16
8.6. FreeRTOS . . . . .	16
8.7. uC/OS . . . . .	16
8.8. Summary . . . . .	16
9. Annex. Changes compared to previous versions . . . . .	17
9.1. Changes between -00 and -01 . . . . .	18

9.2. Changes between -01 and -02 . . . . .	18
9.3. Changes between -02 and -03 . . . . .	18
10. References . . . . .	19
10.1. Normative References . . . . .	19
10.2. Informative References . . . . .	20
Authors' Addresses . . . . .	23

## 1. Introduction

The Internet Protocol suite is being used for connecting Constrained-Node Networks (CNNs) to the Internet, enabling the so-called Internet of Things (IoT) [RFC7228]. In order to meet the requirements that stem from CNNs, the IETF has produced a suite of new protocols specifically designed for such environments (see e.g. [I-D.ietf-lwig-energy-efficient]). New IETF protocol stack components include the IPv6 over Low-power Wireless Personal Area Networks (6LoWPAN) adaptation layer, the IPv6 Routing Protocol for Low-power and lossy networks (RPL) routing protocol, and the Constrained Application Protocol (CoAP).

As of the writing, the main current transport layer protocols in IP-based IoT scenarios are UDP and TCP. However, TCP has been criticized (often, unfairly) as a protocol for the IoT. In fact, some TCP features are not optimal for IoT scenarios, such as relatively long header size, unsuitability for multicast, and always-confirmed data delivery. However, many typical claims on TCP unsuitability for IoT (e.g. a high complexity, connection-oriented approach incompatibility with radio duty-cycling, and spurious congestion control activation in wireless links) are not valid, can be solved, or are also found in well accepted IoT end-to-end reliability mechanisms (see [IntComp] for a detailed analysis).

At the application layer, CoAP was developed over UDP [RFC7252]. However, the integration of some CoAP deployments with existing infrastructure is being challenged by middleboxes such as firewalls, which may limit and even block UDP-based communications. This the main reason why a CoAP over TCP specification has been developed [RFC8323].

Other application layer protocols not specifically designed for CNNs are also being considered for the IoT space. Some examples include HTTP/2 and even HTTP/1.1, both of which run over TCP by default [RFC7540] [RFC2616], and the Extensible Messaging and Presence Protocol (XMPP) [RFC6120]. TCP is also used by non-IETF application-layer protocols in the IoT space such as the Message Queue Telemetry Transport (MQTT) and its lightweight variants.

TCP is a sophisticated transport protocol that includes optional functionality (e.g. TCP options) that may improve performance in some environments. However, many optional TCP extensions require complex logic inside the TCP stack and increase the codesize and the RAM requirements. Many TCP extensions are not required for interoperability with other standard-compliant TCP endpoints. Given the limited resources on constrained devices, careful "tuning" of the TCP implementation can make an implementation more lightweight.

This document provides guidance on how to implement and use TCP in CNNs. The overarching goal is to offer simple measures to allow for lightweight TCP implementation and suitable operation in such environments. A TCP implementation following the guidance in this document is intended to be compatible with a TCP endpoint that is compliant to the TCP standards, albeit possibly with a lower performance. This implies that such a TCP client would always be able to connect with a standard-compliant TCP server, and a corresponding TCP server would always be able to connect with a standard-compliant TCP client.

This document assumes that the reader is familiar with TCP. A comprehensive survey of the TCP standards can be found in [RFC7414]. Similar guidance regarding the use of TCP in special environments has been published before, e.g., for cellular wireless networks [RFC3481].

## 2. Conventions used in this document

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

## 3. Characteristics of CNNs relevant for TCP

### 3.1. Network and link properties

CNNs are defined in [RFC7228] as networks whose characteristics are influenced by being composed of a significant portion of constrained nodes. The latter are characterized by significant limitations on processing, memory, and energy resources, among others [RFC7228]. The first two dimensions pose constraints on the complexity and on the memory footprint of the protocols that constrained nodes can support. The latter requires techniques to save energy, such as radio duty-cycling in wireless devices [I-D.ietf-lwig-energy-efficient], as well as minimization of the number of messages transmitted/received (and their size).

[RFC7228] lists typical network constraints in CNN, including low achievable bitrate/throughput, high packet loss and high variability of packet loss, highly asymmetric link characteristics, severe penalties for using larger packets, limits on reachability over time, etc. CNN may use wireless or wired technologies (e.g., Power Line Communication), and the transmission rates are typically low (e.g. below 1 Mbps).

For use of TCP, one challenge is that not all technologies in CNN may be aligned with typical Internet subnetwork design principles [RFC3819]. For instance, constrained nodes often use physical/link layer technologies that have been characterized as 'lossy', i.e., exhibit a relatively high bit error rate. Dealing with corruption loss is one of the open issues in the Internet [RFC6077].

### 3.2. Usage scenarios

There are different deployment and usage scenarios for CNNs. Some CNNs follow the star topology, whereby one or several hosts are linked to a central device that acts as a router connecting the CNN to the Internet. CNNs may also follow the multihop topology [RFC6606]. One key use case for the use of TCP is a model where constrained devices connect to unconstrained servers in the Internet. But it is also possible that both TCP endpoints run on constrained devices.

In constrained environments, there can be different types of devices [RFC7228]. For example, there can be devices with single combined send/receive buffer, devices with a separate send and receive buffer, or devices with a pool of multiple send/receive buffers. In the latter case, it is possible that buffers also be shared for other protocols.

When a CNN comprising one or more constrained devices and an unconstrained device communicate over the Internet using TCP, the communication possibly has to traverse a middlebox (e.g. a firewall, NAT, etc.). Figure 1 illustrates such scenario. Note that the scenario is asymmetric, as the unconstrained device will typically not suffer the severe constraints of the constrained device. The unconstrained device is expected to be mains-powered, to have high amount of memory and processing power, and to be connected to a resource-rich network.

Assuming that a majority of constrained devices will correspond to sensor nodes, the amount of data traffic sent by constrained devices (e.g. sensor node measurements) is expected to be higher than the amount of data traffic in the opposite direction. Nevertheless, constrained devices may receive requests (to which they may respond),

commands (for configuration purposes and for constrained devices including actuators) and relatively infrequent firmware/software updates.

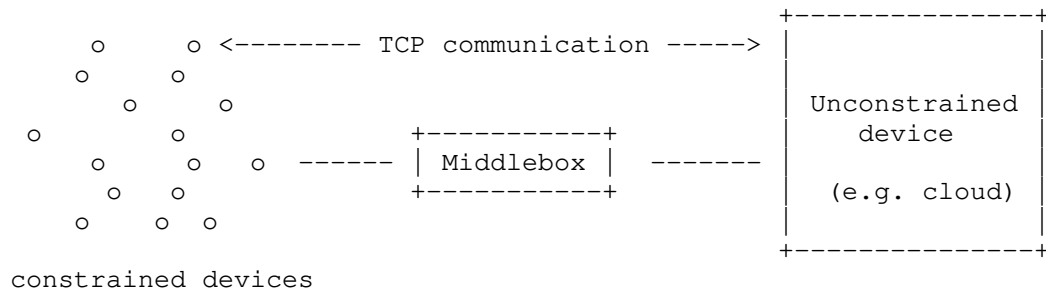


Figure 1: TCP communication between a constrained device and an unconstrained device, traversing a middlebox.

### 3.3. Communication and traffic patterns

IoT applications are characterized by a number of different communication patterns. The following non-comprehensive list explains some typical examples:

- o Unidirectional transfers: An IoT device (e.g. a sensor) can send (repeatedly) updates to the other endpoint. Not in every case there is a need for an application response back to the IoT device.
- o Request-response patterns: An IoT device receiving a request from the other endpoint, which triggers a response from the IoT device.
- o Bulk data transfers: A typical example for a long file transfer would be an IoT device firmware update.

A typical communication pattern is that a constrained device communicates with an unconstrained device (cf. Figure 1). But it is also possible that constrained devices communicate amongst themselves.

## 4. TCP implementation and configuration in CNNs

This section explains how a TCP stack can deal with typical constraints in CNN. The guidance in this section relates to the TCP implementation and its configuration.

#### 4.1. Path properties

##### 4.1.1. Maximum Segment Size (MSS)

Some link layer technologies in the CNN space are characterized by a short data unit payload size, e.g. up to a few tens or hundreds of bytes. For example, the maximum frame size in IEEE 802.15.4 is 127 bytes. 6LoWPAN defined an adaptation layer to support IPv6 over IEEE 802.15.4 networks. The adaptation layer includes a fragmentation mechanism, since IPv6 requires the layer below to support an MTU of 1280 bytes [RFC2460], while IEEE 802.15.4 lacked fragmentation mechanisms. 6LoWPAN defines an IEEE 802.15.4 link MTU of 1280 bytes [RFC4944]. Other technologies, such as Bluetooth LE [RFC7668], ITU-T G.9959 [RFC7428] or DECT-ULE [RFC8105], also use 6LoWPAN-based adaptation layers in order to enable IPv6 support. These technologies do support link layer fragmentation. By exploiting this functionality, the adaptation layers that enable IPv6 over such technologies also define an MTU of 1280 bytes.

On the other hand, there exist technologies also used in the CNN space, such as Master Slave / Token Passing (TP) [RFC8163], Narrowband IoT (NB-IoT) [I-D.ietf-lpwan-overview] or IEEE 802.11ah [I-D.delcarpio-6lo-wlanah], that do not suffer the same degree of frame size limitations as the technologies mentioned above. The MTU for MS/TP is recommended to be 1500 bytes [RFC8163], the MTU in NB-IoT is 1600 bytes, and the maximum frame payload size for IEEE 802.11ah is 7991 bytes.

For the sake of lightweight implementation and operation, unless applications require handling large data units (i.e. leading to an IPv6 datagram size greater than 1280 bytes), it may be desirable to limit the MTU to 1280 bytes in order to avoid the need to support Path MTU Discovery [RFC1981].

An IPv6 datagram size exceeding 1280 bytes can be avoided by setting the TCP MSS not larger than 1220 bytes. (Note: IP version 6 is assumed.)

##### 4.1.2. Explicit Congestion Notification (ECN)

Explicit Congestion Notification (ECN) [RFC3168] has a number of benefits that are relevant for CNNs. ECN allows a router to signal in the IP header of a packet that congestion is arising, for example when a queue size reaches a certain threshold. An ECN-enabled TCP receiver will echo back the congestion signal to the TCP sender by setting a flag in its next TCP ACK. The sender triggers congestion control measures as if a packet loss had happened. ECN can be incrementally deployed in the Internet. Guidance on configuration

and usage of ECN is provided in [RFC7567]. The document [RFC8087] outlines the principal gains in terms of increased throughput, reduced delay, and other benefits when ECN is used over a network path that includes equipment that supports Congestion Experienced (CE) marking.

ECN can reduce packet losses since congestion control measures can be applied earlier [RFC2884]. Less lost packets implies that the number of retransmitted segments decreases, which is particularly beneficial in CNNs, where energy and bandwidth resources are typically limited. Also, it makes sense to try to avoid packet drops for transactional workloads with small data sizes, which are typical for CNNs. In such traffic patterns, it is more difficult to detect packet loss without retransmission timeouts (e.g., as there may be no three duplicate ACKs). Any retransmission timeout slows down the data transfer significantly. When the congestion window of a TCP sender has a size of one segment, the TCP sender resets the retransmit timer, and the sender will only be able to send a new packet when the retransmit timer expires [RFC3168]. Effectively, the TCP sender reduces at that moment its sending rate from 1 segment per Round Trip Time (RTT) to 1 segment per RTO, which can result in a very low throughput. In addition to better throughput, ECN can also help reducing latency and jitter.

Given the benefits, more and more TCP stacks in the Internet support ECN, and it specifically makes sense to leverage ECN in controlled environments such as CNNs.

#### 4.1.3. Explicit loss notifications

There has been a significant body of research on solutions capable of explicitly indicating whether a TCP segment loss is due to corruption, in order to avoid activation of congestion control mechanisms [ETEN] [RFC2757]. While such solutions may provide significant improvement, they have not been widely deployed and remain as experimental work. In fact, as of today, the IETF has not standardized any such solution.

#### 4.2. TCP guidance for small windows and buffers

This section discusses TCP stacks that focus on transferring a single MSS. More general guidance is provided in Section 4.3.

##### 4.2.1. Single-MSS stacks - benefits and issues

A TCP stack can reduce the RAM requirements by advertising a TCP window size of one MSS, and also transmit at most one MSS of unacknowledged data. In that case, both congestion and flow control



implementation is quite simple. Such a small receive and send window may be sufficient for simple message exchanges in the CNN space. However, only using a window of one MSS can significantly affect performance. A stop-and-wait operation results in low throughput for transfers that exceed the lengths of one MSS, e.g., a firmware download.

If CoAP is used over TCP with the default setting for NSTART in [RFC7252], a CoAP endpoint is not allowed to send a new message to a destination until a response for the previous message sent to that destination has been received. This is equivalent to an application-layer window size of 1. For this use of CoAP, a maximum TCP window of one MSS will be sufficient.

#### 4.2.2. TCP options for single-MSS stacks

A TCP implementation needs to support options 0, 1 and 2 [RFC0793]. These options are sufficient for interoperability with a standard-compliant TCP endpoint, albeit many TCP stacks support additional options and can negotiate their use.

A TCP implementation for a constrained device that uses a single-MSS TCP receive or transmit window size may not benefit from supporting the following TCP options: Window scale [RFC1323], TCP Timestamps [RFC1323], Selective Acknowledgments (SACK) and SACK-Permitted [RFC2018]. Also other TCP options may not be required on a constrained device with a very lightweight implementation.

One potentially relevant TCP option in the context of CNNs is TCP Fast Open (TFO) [RFC7413]. As described in Section 5.2.2, TFO can be used to address the problem of traversing middleboxes that perform early filter state record deletion.

#### 4.2.3. Delayed Acknowledgments for single-MSS stacks

TCP Delayed Acknowledgments are meant to reduce the number of transferred bytes within a TCP connection, but they may increase the time until a sender may receive an ACK. There can be interactions with stacks that use very small windows.

A device that advertises a single-MSS receive window should avoid use of delayed ACKs in order to avoid contributing unnecessary delay (of up to 500 ms) to the RTT [RFC5681], which limits the throughput and can increase the data delivery time.

A device that can send at most one MSS of data is significantly affected if the receiver uses delayed ACKs, e.g., if a TCP server or receiver is outside the CNN. One known workaround is to split the

data to be sent into two segments of smaller size. A standard compliant TCP receiver will then immediately acknowledge the second segment, which can improve throughput. This "split hack" works if the TCP receiver uses Delayed Acks, but the downside is the overhead of sending two IP packets instead of one.

#### 4.2.4. RTO estimation for single-MSS stacks

The Retransmission Timeout (RTO) estimation is one of the fundamental TCP algorithms. There is a fundamental trade-off: A short, aggressive RTO behavior reduces wait time before retransmissions, but it also increases the probability of spurious timeouts. The latter lead to unnecessary waste of potentially scarce resources in CNNs such as energy and bandwidth. In contrast, a conservative timeout can result in long error recovery times and thus needlessly delay data delivery.

[RFC6298] describes the standard TCP RTO algorithm. If a TCP sender uses very small window size and cannot use Fast Retransmit/Fast Recovery or SACK, the Retransmission Timeout (RTO) algorithm has a larger impact on performance than for a more powerful TCP stack. In that case, RTO algorithm tuning may be considered, although careful assessment of possible drawbacks is recommended.

As an example, an adaptive RTO algorithm for CoAP over UDP has been defined [I-D.ietf-core-cocoa] that has been found to perform well in CNN scenarios [Commag].

#### 4.3. General recommendations for TCP in CNNs

This section summarizes some widely used techniques to improve TCP, with a focus on their use in CNNs. The TCP extensions discussed here are useful in a wide range of network scenarios, including CNNs. This section is not comprehensive. A comprehensive survey of TCP extensions is published in [RFC7414].

##### 4.3.1. Error recovery and congestion/flow control

Devices that have enough memory to allow larger TCP window size can leverage a more efficient error recovery using Fast Retransmit and Fast Recovery [RFC5681]. These algorithms work efficiently for window sizes of at least 5 MSS: If in a given TCP transmission of segments 1,2,3,4,5, and 6 the segment 2 gets lost, the sender should get an acknowledgement for segment 1 when 3 arrives and duplicate acknowledgements when 4, 5, and 6 arrive. It will retransmit segment 2 when the third duplicate ack arrives. In order to have segment 2, 3, 4, 5, and 6 sent, the window has to be at least five. With an MSS of 1220 byte, a buffer of the size of 5 MSS would require 6100 byte.

For bulk data transfers further TCP improvements may also be useful, such as limited transmit [RFC3042].

#### 4.3.2. Selective Acknowledgments (SACK)

If a device with less severe memory and processing constraints can afford advertising a TCP window size of several MSSs, it makes sense to support the SACK option to improve performance. SACK allows a data receiver to inform the data sender of non-contiguous data blocks received, thus a sender (having previously sent the SACK-Permitted option) can avoid performing unnecessary retransmissions, saving energy and bandwidth, as well as reducing latency. SACK is particularly useful for bulk data transfers. The receiver supporting SACK will need to manage the reception of possible out-of-order received segments, requiring sufficient buffer space. SACK adds  $8*n+2$  bytes to the TCP header, where  $n$  denotes the number of data blocks received, up to 4 blocks. For a low number of out-of-order segments, the header overhead penalty of SACK is compensated by avoiding unnecessary retransmissions.

#### 4.3.3. Delayed Acknowledgments

For certain traffic patterns, Delayed Acknowledgements may have a detrimental effect, as already noted in Section 4.2.3. Advanced TCP stacks may use heuristics to determine the maximum delay for an ACK. For CNNs, the recommendation depends on the expected communication patterns.

If a stack is able to deal with more than one MSS of data, it may make sense to use a small timeout or disable delayed ACKs when traffic over a CNN is expected to mostly be small messages with a size typically below one MSS. For request-response traffic between a constrained device and a peer (e.g. backend infrastructure) that uses delayed ACKs, the maximum ACK rate of the peer will be typically of one ACK every 200 ms (or even lower). If in such conditions the peer device is administered by the same entity managing the constrained device, it is recommended to disable delayed ACKs at the peer side.

In contrast, delayed ACKs allow to reduce the number of ACKs in bulk transfer type of traffic, e.g. for firmware/software updates or for transferring larger data units containing a batch of sensor readings.

### 5. TCP usage recommendations in CNNs

This section discusses how a TCP stack can be used by applications that are developed for CNN scenarios. These remarks are by and large independent of how TCP is exactly implemented.

### 5.1. TCP connection initiation

In the constrained device to unconstrained device scenario illustrated above, a TCP connection is typically initiated by the constrained device, in order for this device to support possible sleep periods to save energy.

### 5.2. TCP connection lifetime

[[TODO: This section may need rewording in the next revision.]]

#### 5.2.1. Long TCP connection lifetime

In CNNs, in order to minimize message overhead, a TCP connection should be kept open as long as the two TCP endpoints have more data to exchange or it is envisaged that further segment exchanges will take place within an interval of two hours since the last segment has been sent. A greater interval may be used in scenarios where applications exchange data infrequently.

TCP keep-alive messages [RFC1122] may be supported by a server, to check whether a TCP connection is active, in order to release state of inactive connections. This may be useful for servers running on memory-constrained devices.

Since the keep-alive timer may not be set to a value lower than two hours [RFC1122], TCP keep-alive messages are not useful to guarantee that filter state records in middleboxes such as firewalls will not be deleted after an inactivity interval typically in the order of a few minutes [RFC6092]. In scenarios where such middleboxes are present, alternative measures to avoid early deletion of filter state records (which might lead to frequent establishment of new TCP connections between the two involved endpoints) include increasing the initial value for the filter state inactivity timers (if possible), and using application layer heartbeat messages.

#### 5.2.2. Short TCP connection lifetime

A different approach to addressing the problem of traversing middleboxes that perform early filter state record deletion relies on using TFO [RFC7413]. In this case, instead of trying to maintain a TCP connection for long time, possibly short-lived connections can be opened between two endpoints while incurring low overhead. In fact, TFO allows data to be carried in SYN (and SYN-ACK) packets, and to be consumed immediately by the receiving endpoint, thus reducing overhead compared with the traditional three-way handshake required to establish a TCP connection.

For security reasons, TFO requires the TCP endpoint that will open the TCP connection (which in CNNs will typically be the constrained device) to request a cookie from the other endpoint. The cookie, with a size of 4 or 16 bytes, is then included in SYN packets of subsequent connections. The cookie needs to be refreshed (and obtained by the client) after a certain amount of time. Nevertheless, TFO is more efficient than frequently opening new TCP connections (by using the traditional three-way handshake) for transmitting new data, as long as the cookie update rate is well below the data new connection rate.

### 5.3. Number of parallel connections

[[TODO: This has been added in -02 but needs further alignment]]

TCP endpoints with a small amount of RAM may only support a small number of connections. Each connection may result in overhead, and depending on the internal TCP implementation, they may compete for scarce resources. A careful application design may try to keep the number of parallel connections as small as possible.

## 6. Security Considerations

Best current practise for securing TCP and TCP-based communication also applies to CNN. As example, use of Transport Layer Security (TLS) is strongly recommended if it is applicable.

There are also TCP options which can improve TCP security. Examples include the TCP MD5 signature option [RFC2385] and the TCP Authentication Option (TCP-AO) [RFC5925]. However, both options add overhead and complexity. The TCP MD5 signature option adds 18 bytes to every segment of a connection. TCP-AO typically has a size of 16-20 bytes.

For the mechanisms discussed in this document, the corresponding considerations apply. For instance, if TFO is used, the security considerations of [RFC7413] apply.

Constrained devices are expected to support smaller TCP window sizes than less limited devices. In such conditions, segment retransmission triggered by RTO expiration is expected to be relatively frequent, due to lack of (enough) duplicate ACKs, especially when a constrained device uses a single-MSS window size. For this reason, constrained devices running TCP may appear as particularly appealing victims of the so-called "shrew" Denial of Service (DoS) attack [shrew], whereby one or more sources generate a packet spike targetted to coincide with consecutive RTO-expiration-triggered retry attempts of a victim node. Note that the attack may

be performed by Internet-connected devices, including constrained devices in the same CNN as the victim, as well as remote ones. Mitigation techniques include RTO randomization and attack blocking by routers able to detect shrew attacks based on their traffic pattern.

## 7. Acknowledgments

Carles Gomez has been funded in part by the Spanish Government (Ministerio de Educacion, Cultura y Deporte) through the Jose Castillejo grant CAS15/00336 and by European Regional Development Fund (ERDF) and the Spanish Government through project TEC2016-79988-P, AEI/FEDER, UE. Part of his contribution to this work has been carried out during his stay as a visiting scholar at the Computer Laboratory of the University of Cambridge.

The authors appreciate the feedback received for this document. The following folks provided comments that helped improve the document: Carsten Bormann, Zhen Cao, Wei Genyu, Ari Keranen, Abhijan Bhattacharyya, Andres Arcia-Moret, Yoshifumi Nishida, Joe Touch, Fred Baker, Nik Sultana, Kerry Lynn, Erik Nordmark, Markku Kojo, and Hannes Tschofenig. Simon Brummer provided details, and kindly performed RAM and ROM usage measurements, on the RIOT TCP implementation. Xavi Vilajosana provided details on the OpenWSN TCP implementation. Rahul Jadhav provided details on the uIP TCP implementation.

## 8. Annex. TCP implementations for constrained devices

This section overviews the main features of TCP implementations for constrained devices. The survey is limited to open source stacks with small footprint. It is not meant to be all-encompassing. For more powerful embedded systems (e.g., with 32-bit processors), there are further stacks that comprehensively implement TCP. On the other hand, please be aware that this Annex is based on information available as of the writing.

### 8.1. uIP

uIP is a TCP/IP stack, targetted for 8 and 16-bit microcontrollers, which pioneered TCP/IP implementations for constrained devices. uIP has been deployed with Contiki and the Arduino Ethernet shield. A code size of ~5 kB (which comprises checksumming, IP, ICMP and TCP) has been reported for uIP [Dunk].

uIP uses same buffer both incoming and outgoing traffic, with has a size of a single packet. In case of a retransmission, an application

must be able to reproduce the same user data that had been transmitted.

The MSS is announced via the MSS option on connection establishment and the receive window size (of one MSS) is not modified during a connection. Stop-and-wait operation is used for sending data. Among other optimizations, this allows to avoid sliding window operations, which use 32-bit arithmetic extensively and are expensive on 8-bit CPUs.

Contiki uses the "split hack" technique (see Section 4.2.3) to avoid delayed ACKs for senders using a single MSS.

## 8.2. lwIP

lwIP is a TCP/IP stack, targetted for 8- and 16-bit microcontrollers. lwIP has a total code size of ~14 kB to ~22 kB (which comprises memory management, checksumming, network interfaces, IP, ICMP and TCP), and a TCP code size of ~9 kB to ~14 kB [Dunk].

In contrast with uIP, lwIP decouples applications from the network stack. lwIP supports a TCP transmission window greater than a single segment, as well as buffering of incoming and outgoing data. Other implemented mechanisms comprise slow start, congestion avoidance, fast retransmit and fast recovery. SACK and Window Scale have been recently added to lwIP.

## 8.3. RIOT

The RIOT TCP implementation (called GNRC TCP) has been designed for Class 1 devices [RFC 7228]. The main target platforms are 8- and 16-bit microcontrollers. GNRC TCP offers a similar function set as uIP, but it provides and maintains an independent receive buffer for each connection. In contrast to uIP, retransmission is also handled by GNRC TCP. GNRC TCP uses a single-MSS window size, which simplifies the implementation. The application programmer does not need to know anything about the TCP internals, therefore GNRC TCP can be seen as a user-friendly uIP TCP implementation.

The MSS is set on connections establishment and cannot be changed during connection lifetime. GNRC TCP allows multiple connections in parallel, but each TCB must be allocated somewhere in the system. By default there is only enough memory allocated for a single TCP connection, but it can be increased at compile time if the user needs multiple parallel connections.

The RIOT TCP implementation does not currently support classic POSIX sockets. However, it supports an interface that has been inspired by POSIX.

#### 8.4. OpenWSN

The TCP implementation in OpenWSN has similar functionality like the uIP TCP implementation. OpenWSN TCP implementation only supports the minimum state machine functionality required. For example, it does not perform retransmissions. There has not been much recent activity to overcome these limitations.

#### 8.5. TinyOS

TinyOS was important as platform for early constrained devices. TinyOS has an experimental TCP stack that uses a simple nonblocking library-based implementation of TCP, which provides a subset of the socket interface primitives. The application is responsible for buffering. The TCP library does not do any receive-side buffering. Instead, it will immediately dispatch new, in-order data to the application and otherwise drop the segment. A send buffer is provided so that the TCP implementation can automatically retransmit missing segments. Multiple TCP connections are possible. Recently there has been little further work on the stack.

#### 8.6. FreeRTOS

FreeRTOS is a real-time operating system kernel for embedded devices that is supported by 16- and 32-bit microprocessors. Its TCP implementation is based on multiple-MSS window size, although a 'Tiny-TCP' option, which is a single-MSS variant, can be enabled. Delayed ACKs are supported, with a 20-ms Delayed ACK timer as a technique intended 'to gain performance'.

#### 8.7. uC/OS

uC/OS is a real-time operating system kernel for embedded devices, which is maintained by Micrium. uC/OS is intended for 8-, 16- and 32-bit microprocessors. The uC/OS TCP implementation supports a multiple-MSS window size.

#### 8.8. Summary



		uIP	lwIP orig	lwIP 2.0	RIOT	OpenWSN	TinyOS	FreeRTOS	uC/	
OS										
A	Memory	Code size(kB)	<5	~9 to ~14	~40	<7	N/A	N/A	<9.2	N/
			(a)	(T1)	(b)	(T3)			(T2)	
t.	Win size(MSS)	1	Mult.	Mult.	1	1	Mult.	Mult.	Mul	
	Slow start	No	Yes	Yes	No	No	Yes	No	Ye	
s	Fast rec/retx	No	Yes	Yes	No	No	Yes	No	Ye	
	Keep-alive	No	No	Yes	No	No	No	Yes	Ye	
o	Win. Scale	No	No	Yes	No	No	No	Yes	N	
	TCP timest.	No	No	Yes	No	No	No	Yes	N	
o	SACK	No	No	Yes	No	No	No	Yes	N	
	Del. ACKs	No	Yes	Yes	No	No	No	Yes	Ye	
s	Socket	No	No	Optional	(I)	Yes	Subset	Yes	Ye	
	Concur. Conn.	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Ye	

(T1) = TCP-only, on x86 and AVR platforms

(T2) = TCP-only, on ARM Cortex-M platform

(T3) = TCP-only, on ARM Cortex-M0+ platform (NOTE: RAM usage for the same platform is ~2.5 kB for one TCP connection plus ~1.2 kB for each additional connection)

(a) = includes IP, ICMP and TCP on x86 and AVR platforms

(b) = the whole protocol stack on mbed

(I) = interface inspired by POSIX

Mult. = Multiple

N/A = Not Available

Figure 2: Summary of TCP features for different lightweight TCP implementations. None of the implementations considered in this Annex support ECN or TFO.

9. Annex. Changes compared to previous versions

RFC Editor: To be removed prior to publication

### 9.1. Changes between -00 and -01

- o Changed title and abstract
- o Clarification that communication with standard-compliant TCP endpoints is required, based on feedback from Joe Touch
- o Additional discussion on communication patterns
- o Numerous changes to address a comprehensive review from Hannes Tschofenig
- o Reworded security considerations
- o Additional references and better distinction between normative and informative entries
- o Feedback from Rahul Jadhav on the uIP TCP implementation
- o Basic data for the TinyOS TCP implementation added, based on source code analysis

### 9.2. Changes between -01 and -02

- o Added text to the Introduction section, and a reference, on traditional bad perception of TCP for IoT
- o Added sections on FreeRTOS and uC/OS
- o Updated TinyOS section
- o Updated summary table
- o Reorganized Section 4 (single-MSS vs multiple-MSS window size), some content now also in new Section 5

### 9.3. Changes between -02 and -03

- o Rewording to better explain the benefit of ECN
- o Additional context information on the surveyed implementations
- o Added details, but removed "Data size" row, in the summary table
- o Added discussion on shrew attacks

## 10. References

### 10.1. 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>>.
- [RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <<https://www.rfc-editor.org/info/rfc1122>>.
- [RFC1323] Jacobson, V., Braden, R., and D. Borman, "TCP Extensions for High Performance", RFC 1323, DOI 10.17487/RFC1323, May 1992, <<https://www.rfc-editor.org/info/rfc1323>>.
- [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>>.
- [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>>.
- [RFC2460] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, DOI 10.17487/RFC2460, December 1998, <<https://www.rfc-editor.org/info/rfc2460>>.
- [RFC3042] Allman, M., Balakrishnan, H., and S. Floyd, "Enhancing TCP's Loss Recovery Using Limited Transmit", RFC 3042, DOI 10.17487/RFC3042, January 2001, <<https://www.rfc-editor.org/info/rfc3042>>.
- [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>>.
- [RFC3819] Karn, P., Ed., Bormann, C., Fairhurst, G., Grossman, D., Ludwig, R., Mahdavi, J., Montenegro, G., Touch, J., and L. Wood, "Advice for Internet Subnetwork Designers", BCP 89, RFC 3819, DOI 10.17487/RFC3819, July 2004, <<https://www.rfc-editor.org/info/rfc3819>>.

- [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>>.
- [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>>.
- [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>>.
- [RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for Constrained-Node Networks", RFC 7228, DOI 10.17487/RFC7228, May 2014, <<https://www.rfc-editor.org/info/rfc7228>>.
- [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>>.

## 10.2. Informative References

- [Commag] A. Betzler, C. Gomez, I. Demirkol, J. Paradells, "CoAP Congestion Control for the Internet of Things", IEEE Communications Magazine, June 2016.
- [Dunk] A. Dunkels, "Full TCP/IP for 8-Bit Architectures", 2003.
- [ETEN] R. Krishnan et al, "Explicit transport error notification (ETEN) for error-prone wireless and satellite networks", Computer Networks 2004.
- [I-D.delcarpio-6lo-wlanah] Vega, I., Robles, I., and R. Morabito, "IPv6 over 802.11ah", draft-delcarpio-6lo-wlanah-01 (work in progress), October 2015.
- [I-D.ietf-core-coap-tcp-tls] Bormann, C., Lemay, S., Tschofenig, H., Hartke, K., Silverajan, B., and B. Raymor, "CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets", draft-ietf-core-coap-tcp-tls-11 (work in progress), December 2017.

- [I-D.ietf-core-cocoa]  
Bormann, C., Betzler, A., Gomez, C., and I. Demirkol,  
"CoAP Simple Congestion Control/Advanced", draft-ietf-  
core-cocoa-03 (work in progress), February 2018.
- [I-D.ietf-lpwan-overview]  
Farrell, S., "LPWAN Overview", draft-ietf-lpwan-  
overview-10 (work in progress), February 2018.
- [I-D.ietf-lwig-energy-efficient]  
Gomez, C., Kovatsch, M., Tian, H., and Z. Cao, "Energy-  
Efficient Features of Internet of Things Protocols",  
draft-ietf-lwig-energy-efficient-08 (work in progress),  
October 2017.
- [IntComp] C. Gomez, A. Arcia-Moret, J. Crowcroft, "TCP in the  
Internet of Things: from ostracism to prominence", IEEE  
Internet Computing, January-February 2018.
- [RFC1981] McCann, J., Deering, S., and J. Mogul, "Path MTU Discovery  
for IP version 6", RFC 1981, DOI 10.17487/RFC1981, August  
1996, <<https://www.rfc-editor.org/info/rfc1981>>.
- [RFC2385] Heffernan, A., "Protection of BGP Sessions via the TCP MD5  
Signature Option", RFC 2385, DOI 10.17487/RFC2385, August  
1998, <<https://www.rfc-editor.org/info/rfc2385>>.
- [RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H.,  
Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext  
Transfer Protocol -- HTTP/1.1", RFC 2616,  
DOI 10.17487/RFC2616, June 1999,  
<<https://www.rfc-editor.org/info/rfc2616>>.
- [RFC2757] Montenegro, G., Dawkins, S., Kojo, M., Magret, V., and N.  
Vaidya, "Long Thin Networks", RFC 2757,  
DOI 10.17487/RFC2757, January 2000,  
<<https://www.rfc-editor.org/info/rfc2757>>.
- [RFC2884] Hadi Salim, J. and U. Ahmed, "Performance Evaluation of  
Explicit Congestion Notification (ECN) in IP Networks",  
RFC 2884, DOI 10.17487/RFC2884, July 2000,  
<<https://www.rfc-editor.org/info/rfc2884>>.
- [RFC3481] Inamura, H., Ed., Montenegro, G., Ed., Ludwig, R., Gurtov,  
A., and F. Khafizov, "TCP over Second (2.5G) and Third  
(3G) Generation Wireless Networks", BCP 71, RFC 3481,  
DOI 10.17487/RFC3481, February 2003,  
<<https://www.rfc-editor.org/info/rfc3481>>.

- [RFC4944] Montenegro, G., Kushalnagar, N., Hui, J., and D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks", RFC 4944, DOI 10.17487/RFC4944, September 2007, <<https://www.rfc-editor.org/info/rfc4944>>.
- [RFC6077] Papadimitriou, D., Ed., Welzl, M., Scharf, M., and B. Briscoe, "Open Research Issues in Internet Congestion Control", RFC 6077, DOI 10.17487/RFC6077, February 2011, <<https://www.rfc-editor.org/info/rfc6077>>.
- [RFC6092] Woodyatt, J., Ed., "Recommended Simple Security Capabilities in Customer Premises Equipment (CPE) for Providing Residential IPv6 Internet Service", RFC 6092, DOI 10.17487/RFC6092, January 2011, <<https://www.rfc-editor.org/info/rfc6092>>.
- [RFC6120] Saint-Andre, P., "Extensible Messaging and Presence Protocol (XMPP): Core", RFC 6120, DOI 10.17487/RFC6120, March 2011, <<https://www.rfc-editor.org/info/rfc6120>>.
- [RFC6606] Kim, E., Kaspar, D., Gomez, C., and C. Bormann, "Problem Statement and Requirements for IPv6 over Low-Power Wireless Personal Area Network (6LoWPAN) Routing", RFC 6606, DOI 10.17487/RFC6606, May 2012, <<https://www.rfc-editor.org/info/rfc6606>>.
- [RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <<https://www.rfc-editor.org/info/rfc7252>>.
- [RFC7414] Duke, M., Braden, R., Eddy, W., Blanton, E., and A. Zimmermann, "A Roadmap for Transmission Control Protocol (TCP) Specification Documents", RFC 7414, DOI 10.17487/RFC7414, February 2015, <<https://www.rfc-editor.org/info/rfc7414>>.
- [RFC7428] Brandt, A. and J. Buron, "Transmission of IPv6 Packets over ITU-T G.9959 Networks", RFC 7428, DOI 10.17487/RFC7428, February 2015, <<https://www.rfc-editor.org/info/rfc7428>>.
- [RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/info/rfc7540>>.

- [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>>.
- [RFC7668] Nieminen, J., Savolainen, T., Isomaki, M., Patil, B., Shelby, Z., and C. Gomez, "IPv6 over BLUETOOTH(R) Low Energy", RFC 7668, DOI 10.17487/RFC7668, October 2015, <<https://www.rfc-editor.org/info/rfc7668>>.
- [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>>.
- [RFC8105] Mariager, P., Petersen, J., Ed., Shelby, Z., Van de Logt, M., and D. Barthel, "Transmission of IPv6 Packets over Digital Enhanced Cordless Telecommunications (DECT) Ultra Low Energy (ULE)", RFC 8105, DOI 10.17487/RFC8105, May 2017, <<https://www.rfc-editor.org/info/rfc8105>>.
- [RFC8163] Lynn, K., Ed., Martocci, J., Neilson, C., and S. Donaldson, "Transmission of IPv6 over Master-Slave/Token-Passing (MS/TP) Networks", RFC 8163, DOI 10.17487/RFC8163, May 2017, <<https://www.rfc-editor.org/info/rfc8163>>.
- [RFC8323] Bormann, C., Lemay, S., Tschofenig, H., Hartke, K., Silverajan, B., and B. Raymor, Ed., "CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets", RFC 8323, DOI 10.17487/RFC8323, February 2018, <<https://www.rfc-editor.org/info/rfc8323>>.
- [shrew] A. Kuzmanovic, E. Knightly, "Low-Rate TCP-Targeted Denial of Service Attacks", SIGCOMM'03 2003.

#### Authors' Addresses

Carles Gomez  
UPC  
C/Esteve Terradas, 7  
Castelldefels 08860  
Spain  
  
Email: [carlesgo@entel.upc.edu](mailto:carlesgo@entel.upc.edu)



Jon Crowcroft  
University of Cambridge  
JJ Thomson Avenue  
Cambridge, CB3 0FD  
United Kingdom

Email: [jon.crowcroft@cl.cam.ac.uk](mailto:jon.crowcroft@cl.cam.ac.uk)

Michael Scharf  
Nokia  
Lorenzstrasse 10  
Stuttgart, 70435  
Germany

Email: [michael.scharf@nokia.com](mailto:michael.scharf@nokia.com)

TCP Maintenance & Minor Extensions (tcpm)  
Internet-Draft  
Intended status: Experimental  
Expires: January 3, 2019

B. Briscoe  
CableLabs  
M. Kuehlewind  
ETH Zurich  
R. Scheffenegger  
July 2, 2018

More Accurate ECN Feedback in TCP  
draft-ietf-tcpm-accurate-ecn-07

Abstract

Explicit Congestion Notification (ECN) is a mechanism where network nodes can mark IP packets instead of dropping them to indicate incipient congestion to the end-points. Receivers with an ECN-capable transport protocol feed back this information to the sender. ECN is specified for TCP in such a way that only one feedback signal can be transmitted per Round-Trip Time (RTT). Recently, new TCP mechanisms like Congestion Exposure (ConEx), 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 an experimental 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. 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 is submitted in full conformance with the provisions of BCP 78 and BCP 79.

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

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

This Internet-Draft will expire on January 3, 2019.

## Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

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

## Table of Contents

1. Introduction . . . . .	3
1.1. Document Roadmap . . . . .	4
1.2. Goals . . . . .	5
1.3. Experiment Goals . . . . .	5
1.4. Terminology . . . . .	6
1.5. Recap of Existing ECN feedback in IP/TCP . . . . .	7
2. AccECN Protocol Overview and Rationale . . . . .	8
2.1. Capability Negotiation . . . . .	9
2.2. Feedback Mechanism . . . . .	9
2.3. Delayed ACKs and Resilience Against ACK Loss . . . . .	10
2.4. Feedback Metrics . . . . .	10
2.5. Generic (Dumb) Reflector . . . . .	11
3. AccECN Protocol Specification . . . . .	12
3.1. Negotiating to use AccECN . . . . .	12
3.1.1. Negotiation during the TCP handshake . . . . .	12
3.1.2. Retransmission of the SYN . . . . .	14
3.2. AccECN Feedback . . . . .	15
3.2.1. Initialization of Feedback Counters at the Data Sender . . . . .	16
3.2.2. The ACE Field . . . . .	16
3.2.3. Testing for Zeroing of the ACE Field . . . . .	18
3.2.4. Testing for Mangling of the IP/ECN Field . . . . .	18
3.2.5. Safety against Ambiguity of the ACE Field . . . . .	19
3.2.6. The AccECN Option . . . . .	20
3.2.7. Path Traversal of the AccECN Option . . . . .	21
3.2.8. Usage of the AccECN TCP Option . . . . .	25
3.3. Requirements for TCP Proxies, Offload Engines and other Middleboxes on AccECN Compliance . . . . .	26
4. Interaction with Other TCP Variants . . . . .	27
4.1. Compatibility with SYN Cookies . . . . .	27
4.2. Compatibility with Other TCP Options and Experiments . . . . .	28

4.3. Compatibility with Feedback Integrity Mechanisms . . . .	28
5. Protocol Properties . . . . .	29
6. IANA Considerations . . . . .	31
7. Security Considerations . . . . .	32
8. Acknowledgements . . . . .	33
9. Comments Solicited . . . . .	33
10. References . . . . .	33
10.1. Normative References . . . . .	33
10.2. Informative References . . . . .	34
Appendix A. Example Algorithms . . . . .	36
A.1. Example Algorithm to Encode/Decode the AcceECN Option . .	36
A.2. Example Algorithm for Safety Against Long Sequences of ACK Loss . . . . .	37
A.2.1. Safety Algorithm without the AcceECN Option . . . . .	37
A.2.2. Safety Algorithm with the AcceECN Option . . . . .	39
A.3. Example Algorithm to Estimate Marked Bytes from Marked Packets . . . . .	40
A.4. Example Algorithm to Beacon AcceECN Options . . . . .	41
A.5. Example Algorithm to Count Not-ECT Bytes . . . . .	42
Appendix B. Rationale for Usage of TCP Header Flags . . . . .	42
B.1. Three TCP Header Flags in the SYN-SYN/ACK Handshake . . .	42
B.2. Four Codepoints in the SYN/ACK . . . . .	43
B.3. Space for Future Evolution . . . . .	44
Authors' Addresses . . . . .	44

## 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 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 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 AcceECN for short. If AcceECN 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 and allocates the currently reserved flag (previously called NS) in the TCP header, to be used as one 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 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 testing confirms that the proposed mechanism can be deployed at large scale. 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 feasibility 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 pattern which can particularly have an impact on hardware offload. It is currently specified as a hard requirement, because the sender can exploit the predictability of the receiver's behaviour. However, further experimentation is needed to advise if it will have to become just preferred 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
Header Length				Reserved			N S	C W R	E C E	U R G	A C K	P S H	R S T	S Y N	F I N

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

## 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 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 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 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 (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 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 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 timeouts). 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.elb`. The CE packet counter (`r.cep`), counts the number of packets the host receives with the CE code point in the IP ECN field, including CE marks on control packets without data. `r.ceb`, `r.e0b` and `r.elb` count the number of TCP payload bytes in packets marked respectively with the CE, ECT(0) and ECT(1) codepoint in their IP-ECN field. When a host first enters AccECN mode, it initializes its counters to `r.cep = 5`, `r.e0b = 1` and `r.ceb = r.elb = 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.elb` 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.elb = 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
Header Length				Reserved			ACE			U R G	A C K	P S H	R S T	S Y N	F I 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 AcceECN 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 AcceECN 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 AcceECN 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 AcceECN 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 AcceECN 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 AcceECN 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 AcceECN is also using a stateless handshake (termed a SYN cookie) it will not remember whether it entered AcceECN 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 3WSH 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:

- o the not-ECT codepoint changes;
- o either ECT codepoint transitions to not-ECT;
- o 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.elb counters, respectively. The initial 'E' of each field name stands for 'Echo'.

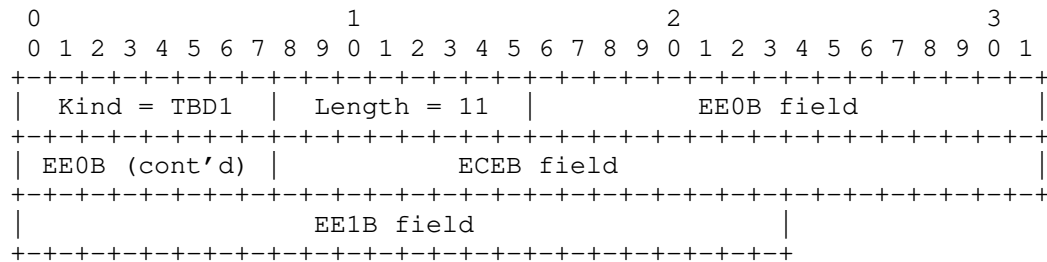


Figure 3: The AcceCN Option

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

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

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

Whenever a Data Receiver sends an AcceCN Option, the rules in Section 3.2.8 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 intends to use the AcceCN option at any time during the rest of the connection it is 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 AcceECN SHOULD include an AcceECN Option in the first ACK at the end of the 3WSH. However, this first ACK is not delivered reliably, so the TCP client SHOULD also include an AcceECN Option on the first data segment it sends (if it ever sends one).

A host MAY NOT include an AcceECN 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 AcceECN Option.

#### 3.2.7.2. Testing for Loss of Packets Carrying the AcceECN 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 AcceECN 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 AcceECN Option. If this retransmission times out, to expedite connection setup, the TCP server SHOULD disable AcceECN and ECN for this connection by retransmitting the SYN/ACK with AE=CWR=ECE=0 and no AcceECN 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 AcceECN 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 AcceECN Option was lost, it SHOULD fall back to no AcceECN Option on the retransmission. Again, implementers MAY use other fall-back strategies such as attempting to retransmit a second segment with the AcceECN Option before fall-back, and/or caching whether the AcceECN Option is blocked for subsequent connections.

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

If the TCP server receives a second SYN with a request for AcceECN support, it should resend the SYN/ACK, again confirming its support for AcceECN, but this time without the AcceECN 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 AcceECN 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 AcceECN 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 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. 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 3WSH so that it can fall back to operation without ECN and/or operation without the AccECN Option.

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

## 6. IANA Considerations

This document 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: 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 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-07 (work in progress), March 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-02 (work in  
progress), March 2018.
- [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:

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

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

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

where `'%'` is the modulo operator.

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

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

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

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

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

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

Two variants of the algorithm are given: i) a more conservative variant for a Data Sender to use if it detects that the AccECN Option is not available (see Section 3.2.5 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:

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

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

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

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

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

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

Section 3.2.5 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 newlyAcedB is the amount of data that an ACK acknowledges since the previous ACK, then the Data Sender could assume that this acknowledges newlyAcedPkt full-sized segments, where newlyAcedPkt = newlyAcedB/MSS. Then it could assume that the ACE field incremented by

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

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

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

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

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

If missing acknowledgement numbers arrive later (due to reordering), Section 3.2.5 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.

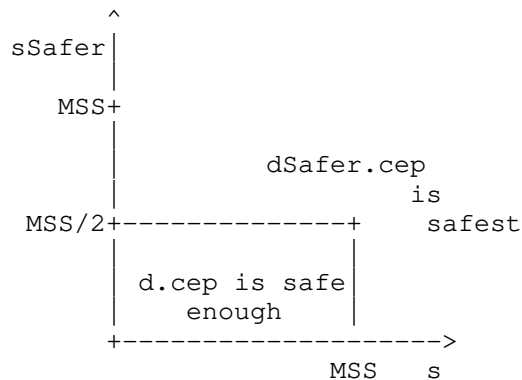
#### A.2.2. Safety Algorithm with the AcceCN Option

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:





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.elb. 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 3WSH.

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

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

### Appendix B. Rationale for Usage of TCP Header Flags

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

AccECN uses a rather unorthodox but justified approach to negotiate the highest version TCP ECN feedback scheme that both ends support. It follows from the original TCP ECN capability negotiation [RFC3168], in which the client set the 2 least significant 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. It is not recorded why ECN originally used this approach instead of the more orthodox use of a TCP option.

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 AcceECN 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 AcceECN 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 AcceECN 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 AcceECN 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 AcceECN, 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, AcceECN 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 AcceECN, an AcceECN server could use only 2 of the 4 remaining codepoints. They both indicated AcceECN 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:** The requirement not to reject unexpected initial values of the ACE counter (in the main TCP header) in the last para of Section 3.2.3 ensures that 5 unused codepoints on the final ACK of the 3-way handshake and 7 unused values on the first data packet from the server could be used to negotiate future variants of the AccECN protocol between the endpoints. Also, a similar requirement not to reject unexpected initial values in the TCP option is for the same purpose. If traversal of the TCP option were reliable, this would have enabled a far wider range of future variation.

**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. All possible combinations of SYN/ACK could be used in response except 000 and reflection of the same values sent on the SYN. These would not allow fall-back to Classic ECN support for a server that did not understand them, but they are available, perhaps for uses other than ECN in future.

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  
CableLabs  
UK

EMail: [ietf@bobbriscoe.net](mailto:ietf@bobbriscoe.net)  
URI: <http://bobbriscoe.net/>

Mirja Kuehlewind  
ETH Zurich  
Zurich  
Switzerland

EMail: [mirja.kuehlewind@tik.ee.ethz.ch](mailto:mirja.kuehlewind@tik.ee.ethz.ch)

Richard Scheffenegger  
Vienna  
Austria

EMail: [rscheff@gmx.at](mailto:rscheff@gmx.at)

TCP Maintenance Working Group  
Internet-Draft  
Intended status: Experimental  
Expires: January 3, 2019

Y. Cheng  
N. Cardwell  
N. Dukkipati  
P. Jha  
Google, Inc  
July 2, 2018

RACK: a time-based fast loss detection algorithm for TCP  
draft-ietf-tcpm-rack-04

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

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

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

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

This Internet-Draft will expire on January 3, 2019.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect

to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## 1. Introduction

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 [RFC4653] [RFC5827] [RFC5681] [RFC6675] [RFC7765] [FACK] [THIN-STREAM], 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 RFC6675 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 more effective algorithm to handle loss and reordering.

## 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][RFC6675][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) alone 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][RFC6675][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. Design Rationale for Reordering Tolerance

The reordering behavior of networks can evolve (over years) in response to the behavior of transport protocols and applications, as well as the needs of network designers and operators. From a network

or link designer's viewpoint, parallelization (eg. link bonding) is the easiest way to get a network to go faster. Therefore their main constraint on speed is reordering, and there is pressure to relax that constraint. If RACK becomes widely deployed, the underlying networks may introduce more reordering for higher throughput. But this may result in excessive reordering that hurts end to end performance:

1. End host packet processing: extreme reordering on high-speed networks would incur high CPU cost by greatly reducing the effectiveness of aggregation mechanisms, such as large receive offload (LRO) and generic receive offload (GRO), and significantly increasing the number of ACKs.
2. Congestion control: TCP congestion control implicitly assumes the feedback from ACKs are from the same bottleneck. Therefore it cannot handle well scenarios where packets are traversing largely disjoint paths.
3. Loss recovery: Having an excessively large reordering window to accommodate widely different latencies from different paths would increase the latency of loss recovery.

An end-to-end transport protocol cannot tell immediately whether a hole is reordering or loss. It can only distinguish between the two in hindsight if the hole in the sequence space gets filled later without a retransmission. How long the sender waits for such potential reordering events to settle is determined by the current reordering window.

Given these considerations, a core design philosophy of RACK is to adapt to the measured duration of reordering events, within reasonable and specific bounds. To accomplish this RACK places the following mandates on the reordering window:

1. The initial RACK reordering window SHOULD be set to a small fraction of the round-trip time.
2. If no reordering has been observed, then RACK SHOULD honor the classic 3-DUPACK rule for initiating fast recovery. One simple way to implement this is to temporarily override the reorder window to 0.
3. The RACK reordering window SHOULD leverage Duplicate Selective Acknowledgement (DSACK) information [RFC3708] to adaptively estimate the duration of reordering events.

4. The RACK reordering window MUST be bounded and this bound SHOULD be one round trip.

As a flow starts, either condition 1 or condition 2 or both would trigger RACK to start the recovery process quickly. The low initial reordering window and use of the 3-DUPACK rule are key to achieving low-latency loss recovery for short flows by risking spurious retransmissions to recover losses quickly. This rationale is that spurious retransmissions for short flows are not expected to produce excessive network traffic.

For long flows the design tolerates reordering within a round trip. This handles reordering caused by path divergence in small time scales (reordering within the round-trip time of the shortest path), which should tolerate much of the reordering from link bonding, multipath routing, or link-layer out-of-order delivery. It also relaxes ordering constraints to allow sending flights of TCP packets on different paths dynamically for better load-balancing (e.g. flowlets).

However, the fact that the initial RACK reordering window is low, and the RACK reordering window's adaptive growth is bounded, means that there will continue to be a cost to reordering and a limit to RACK's adaptation to reordering. This maintains a disincentive for network designers and operators to introduce needless or excessive reordering, particularly because they have to allow for low round trip time paths. This means RACK will not encourage networks to perform inconsiderate fine-grained packet-spraying over highly disjoint paths with very different characteristics. There are good alternative solutions, such as MPTCP, for such networks.

To conclude, the RACK algorithm aims to adapt to small degrees of reordering, quickly recover most losses within one to two round trips, and avoid costly retransmission timeouts (RTOs). 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 longer-running flows.

This document provides a concrete and detailed reordering window adaptation algorithm for implementors. We note that the specifics of the algorithm are likely to evolve over time. But that is a separate engineering optimization that's out of scope for this document.

#### 4. 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.

#### 5. 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 RTT of the most recently transmitted packet that has been delivered (either cumulatively acknowledged or selectively acknowledged) on the connection.

"RACK.rtt\_seq" is the SND.NXT when RACK.rtt is updated.

"RACK.reo\_wnd" is a reordering window 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 a DSACK option has been received since last RACK.reo\_wnd change "RACK.pkts\_sacked" returns the total number of packets selectively acknowledged in the SACK scoreboard.

"RACK.reord" indicates the connection has detected packet reordering event(s)

"RACK.fack" is the highest selectively or cumulatively acknowledged sequence

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.

## 6. Algorithm Details

### 6.1. 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.

## 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 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.

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.

Notice that the second condition above is a heuristic. This heuristic would fail to update RACK stats if the packet is spuriously retransmitted because of a recent minimum RTT decrease (e.g. path change). Consequentially RACK may not detect losses from ACK events and the recovery resorts to the (slower) TLP or RTO timer-based events. However such events should be rare and the connection would pick up the new minimum RTT when the recovery ends to avoid repeated similar failures.

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() - Packet.xmit_ts
    If Packet.retransmitted is TRUE:
      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
```

Step 3: Detect packet reordering

To detect reordering, the sender looks for original data packets being delivered out of order in sequence space. The sender tracks the highest sequence selectively or cumulatively acknowledged in the RACK.fack variable. The name fack stands for the most forward ACK originated from the [FACK] draft. If the ACK selectively or cumulatively acknowledges an unacknowledged and also never retransmitted sequence below RACK.fack, then the corresponding packet has been reordered and RACK.reord is set to 1.

The heuristic above only detects reordering if the re-ordered packet has not yet been retransmitted. This is a major drawback because if RACK has a low reordering window and the network is reordering packets, RACK may falsely retransmit frequently. Consequently RACK may fail to detect reordering to increase the reordering window, because the reordered packets were already (falsely) retransmitted.

DSACK [RFC3708] can help mitigate this issue. The false retransmission would solicit DSACK option in the ACK. Therefore if the ACK has a DSACK option covering some sequence that were both acknowledged and retransmitted, this implies the original packet was reordered but RACK retransmitted the packet too quickly and should set RACK.reord to 1.

```
RACK_detect_reordering():
  For each Packet newly acknowledged cumulatively or selectively:
    If Packet.end_seq > RACK.fack:
      RACK.fack = Packet.end_seq
    Else if Packet.end_seq < RACK.fack AND
      Packet.retransmitted is FALSE:
      RACK.reord = TRUE
  For each Packet covered by the DSACK option:
    If Packet.retransmitted is TRUE:
      RACK.reord = TRUE
```

#### Step 4: 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. This section documents a detailed algorithm following the design rationale section. RACK starts initially with a conservative window of  $\text{min\_RTT}/4$ . If no reordering has been observed, RACK uses RACK.reo\_wnd of 0 during loss recovery, in order to retransmit quickly, or when the number of DUPACKs exceeds the classic DUPACK threshold. The subtle difference of this approach and conventional one [RFC5681][RFC6675] is discussed later in the section of "RACK and TLP discussions".

Further, RACK MAY use DSACK [RFC3708] to adapt the reordering window, to higher degrees of reordering, if DSACK is supported. 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 becomes  $(N+1) * \text{min\_RTT} / 4$ , with an upper-bound of SRTT. The inflated RACK.reo\_wnd would persist for 16 loss recoveries and after which it resets to its starting value,  $\text{min\_RTT} / 4$ .

The following pseudocode implements above algorithm. Note that 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 DSACK option is present:
        RACK.dsack = true

    If SND.UNA < RACK.rtt_seq:
        RACK.dsack = false /* React to DSACK once per round trip */

    If RACK.dsack:
        RACK.reo_wnd_incr += 1
        RACK.dsack = false
        RACK.rtt_seq = SND.NXT
        RACK.reo_wnd_persist = 16 /* Keep window for 16 recoveries */
    Else if exiting loss recovery:
        RACK.reo_wnd_persist -= 1
        If RACK.reo_wnd_persist <= 0:
            RACK.reo_wnd_incr = 1

    If RACK.reordering_seen is FALSE:
        If in loss recovery: /* If in fast or timeout recovery */
            RACK.reo_wnd = 0
            Return
        Else if RACK.pkts_sacked >= RACK.dupthresh:
            RACK.reo_wnd = 0
            return
    RACK.reo_wnd = RACK.min_RTT / 4 * RACK.reo_wnd_incr
    RACK.reo_wnd = min(RACK.reo_wnd, SRTT)

```

Step 5: 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:

$$\text{now} \geq \text{Packet.xmit\_ts} + \text{RACK.reo\_wnd} + (\text{RACK.ack\_ts} - \text{RACK.xmit\_ts})$$

Then  $(\text{RACK.ack\_ts} - \text{RACK.xmit\_ts})$  is just the RTT of the packet we used to set RACK.xmit\_ts, so this reduces to:

$$\text{Packet.xmit\_ts} + \text{RACK.rtt} + \text{RACK.reo\_wnd} - \text{now} \leq 0$$

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, not acknowledged yet:
        If Packet.lost is TRUE or Packet.retransmitted is FALSE:
            Continue /* Lost packet not retransmitted yet */

        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:
                Packet.lost = TRUE
            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 abort on the first packet sent after RACK.time\_ts. This can be implemented by using a separate list sorted in time order. The implementation inserts the packet at 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 because it enables implementations to support high-BDP networks. This optimization is implemented in Linux and sees orders of magnitude improvement in CPU usage on high-speed WAN networks.

### 6.3. Tail Loss Probe: fast recovery for 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 of data is lost.
2. Tail losses of data segments at the end of an application transaction.
3. Tail losses of ACKs at the end of an application transaction.
4. 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.
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.

#### 6.4. 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.

## 6.5. 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].

The TLP algorithm has three phases, which we discuss in turn.

### 6.5.1. 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 the following situations:

1. After transmitting new data that was not itself a TLP probe
2. Upon receiving an ACK that cumulatively acknowledges data
3. Upon receiving a SACK that selectively acknowledges data that was last sent before the segment with `SEG.SEQ=SND.UNA` was last (re)transmitted

A sender should schedule a PTO only if all of the following conditions are met:

1. The connection supports SACK [RFC2018]
2. The connection has no SACKed sequences in the SACK scoreboard
3. The connection is not in loss recovery

If a PTO can be scheduled according to these conditions, the sender should schedule a PTO. If there was a previously scheduled PTO or RTO pending, then that pending PTO or RTO should first be cancelled, and then the new PTO should be scheduled.

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  $2 \times \text{SRTT}$  allows a sender to wait long enough to know that an ACK is overdue. Under normal circumstances, i.e. no losses, an ACK typically arrives in one 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, network delay variations and end-system processing latencies and timer granularities can easily delay ACKs beyond  $2 \times \text{SRTT}$ , 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.

### 6.5.2. Phase 2: Sending a loss probe

When the PTO fires, transmit a probe data segment:

```
TLP_send_probe():
  If an unsent segment exists AND
    the receive window allows new data to be sent:
    Transmit the lowest-sequence unsent segment of up to SMSS
    Increment FlightSize by the size of the newly-sent segment
  Else:
    Retransmit the highest-sequence segment sent so far
  The cwnd remains unchanged
```

When the loss probe is a retransmission, the sender uses the highest-sequence segment sent so far. This is in order 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 any required RACK reordering settling timer to then 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 this using TLP recovery detection (see below).

Note that after transmitting a TLP, the sender MUST arm an RTO timer, and not the PTO timer. This ensures that the sender does not send repeated, back-to-back TLP probes.

### 6.5.3. Phase 3: ACK processing

On each incoming ACK, the sender should check the conditions in Step 1 of Phase 1 to see if it should schedule (or reschedule) the loss probe timer.

### 6.6. 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.

#### 6.6.1. Initializing and resetting state

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

```
TLPRxtOut = false
```

#### 6.6.2. 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 TLP<sub>RxtOut</sub> 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()
}
```

## 7. RACK and TLP discussions

### 7.1. 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.<sub>reo\_wnd</sub> (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.<sub>reo\_wnd</sub> (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 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 algorithms based on sequence counting (e.g., [RFC6675][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.

## 7.2. 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 granularity (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.

### 7.3. Adjusting the reordering window

When the sender detects packet reordering, RACK uses a reordering window of  $\text{min\_rtt} / 4$ . 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.

### 7.4. 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.

#### 7.5. 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  $RTO + 4 \cdot RTT$ , with an ending cwnd of 4 packets due to congestion window validation.

With RACK, a sender would send the TLP after  $2 \cdot \text{RTT}$  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  $2 \cdot \text{RTT} + 4 \cdot \text{RTT}$ , 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 ( $\text{RTO} + 4 \cdot \text{RTT}$  vs  $6 \cdot \text{RTT}$ ) 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.

#### 7.6. 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.

#### 7.7. 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 timestamp options [RFC7323]). For example, the QUIC protocol implements RACK [QUIC-LR].

## 8. 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 6675 on

Group 2: RACK on, TLP off, RFC 6675 on

Group 3: RACK on, TLP on, RFC 6675 on

Group 4: RACK on, TLP on, RFC 6675 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-RTT) 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 [RFC6675].

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 RTT-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 RFC6675 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.

## 9. 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.

## 10. IANA Considerations

This document makes no request of IANA.

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

## 11. 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, and Bob Briscoe contributed to the draft and the implementations in Linux, FreeBSD and QUIC.

## 12. References

### 12.1. 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., Dukkupati, 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.

## 12.2. 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.
- [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] Dukkupati, N., Cardwell, N., Cheng, Y., and M. Mathis, "Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Drops", draft-dukkupati-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 Dukkupati  
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

Network Working Group  
Internet-Draft  
Intended status: Experimental  
Expires: December 22, 2018

Y. Nishida  
GE Global Research  
June 20, 2018

Disabling PAWS When Other Protections Are Available  
draft-nishida-tcpm-disabling-paws-00

Abstract

PAWS provides protection against old duplicated segments caused by wrapped sequence or earlier incarnated connections. One drawback of PAWS is that it requires to place timestamp option in all segments, which consumes 10-12 bytes in the option space of TCP. In addition, since PAWS just checks if timestamps is older or not, the protection logic is not very strong against malicious attacks or cannot work properly in some situations. On the other hand, some other technologies which can provide stronger protections than PAWS are becoming available these days. In this document, we propose to utilize other protection mechanisms as replacements of PAWS when they are available.

Status of This Memo

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

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

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

This Internet-Draft will expire on December 22, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

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

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

## Table of Contents

1. Introduction . . . . .	2
2. Conventions and Terminology . . . . .	3
3. Possible Mechanisms AS Replacements of PAWS . . . . .	3
3.1. TCP Increased Security (tcpinc) . . . . .	3
3.2. Multipath TCP . . . . .	4
3.3. TLS . . . . .	4
4. Duplicates from Earlier Connection Incarnations . . . . .	5
5. Security Considerations . . . . .	5
6. IANA Considerations . . . . .	5
7. References . . . . .	5
7.1. Normative References . . . . .	5
7.2. Informative References . . . . .	6
Author's Address . . . . .	7

## 1. Introduction

PAWS (Protect Against Wrapped Sequences) defined in [RFC7323] is a technique that can identify old duplicate segments in a TCP connection or segments from earlier incarnated connections. PAWS utilizes timestamp option in TCP segments. When both TCP endpoints agree to use PAWS, all segments belong to this connection will have the options, which consumes 10-12 bytes of 40 bytes option space. As recent TCP connections use option space for other TCP extensions such as [RFC2018], [RFC5925] and [RFC6824], this feature tends to be considered as expensive these days.

Timestamp option is also used for RTTM (Round Trip Time Measurement). Gathering many RTT samples from the timestamp in every TCP segment may look useful approach to improve RTO estimations. However, some research results shows taking a few timestamps per RTT can be sufficient [MALLMAN99]. Also, some TCP implementations record the transmission time of each packet. In this case, timestamp option is not necessary to measure RTTs.

The basic idea of PAWS is that a received segment is considered as an old duplicate if the timestamp in it is less than the timestamps recently received on a connection. The timestamp values used in PAWS is 32-bit unsigned integers. Hence, when PAWS compares two timestamp values:  $t_1$ ,  $t_2$ , it regards  $t_2$  as "newer than  $t_1$ " if  $0 < (t_2 - t_1) <$

2<sup>31</sup>, otherwise t2 is considered as "older than t1". This logic presumes timestamp is monotonically increased across connections, however, it can be confused in some cases such as where multiple nodes are behind the same NAT. In addition, if malicious attackers try to cheat the PAWS logic with random timestamp values, there will be 50% of chance for success on each try. This basically means PAWS can hardly contribute to securing TCP connections. On the other hand, several technologies which can be utilized for the same purpose are becoming available.

Based on these observations, we propose to utilize other mechanisms as replacements of PAWS when it is possible. The goal of the proposal in the draft is to provide stronger protections for old duplicated segments while facilitating the use of TCP option space by suppressing timestamp options. Another benefit of the proposal is that it can contribute to facilitating recycling of TCP connections in TIME\_WAIT stat, which will be useful for busy servers.

## 2. Conventions and Terminology

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

## 3. Possible Mechanisms AS Replacements of PAWS

In this section, we present several possible mechanisms that can be used as replacements of PAWS. When these mechanisms are available and are activated in a TCP connection, PAWS can be disabled as it will be redundant. In order to disable PAWS protection, a simple signaling mechanism will be needed as this will require agreement on both ends. We discuss how to utilize these mechanisms and how to disable PAWS safely below.

### 3.1. TCP Increased Security (tcpinc)

Currently, TCP extensions that can provide unauthenticated encryption and integrity protection to TCP streams have been actively discussed in IETF [TCPINC]. As these proposed extensions is based on encryption algorithms, old duplicated segments in the same connection or segments from early connections with the same 4 tuples will easily be identified. In addition, it will be harder for malicious attacks to cheat this protections. To utilize this feature as a replacement of PAWS, we propose to update the encryption negotiation option (TCP-ENO) [I-D.ietf-tcpinc-tcpno] by using 1 bit of global suboption in initial suboption byte. This bit indicates that the end point supports disabling PAWS. After endpoints confirm that both ends support disabling PAWS and encryption negotiation has been

successful, they will disable PAWS and will use timestamp options only for RTTM. However, in case TCP-ENO has failed or either side does not support disabling PAWS, PAWS MUST NOT be disabled if the use of TS option is negotiated.

### 3.2. Multipath TCP

Multipath TCP [RFC6824] can also be used for a replacement of PAWS. Multipath TCP maintains 64 bits sequence number space in the session and use DSS (Data Sequence Signal) option for this purpose in addition to the sequence number field in the TCP header. This DSS option can be served to provide the same protections as PAWS. By checking Data sequence number is DSS option, it can identify old duplicated segments. Because the data sequence number in the option should be exactly matched with the number stored in the session, MPTCP can provide stronger protection than PAWS. One way to signal disabling PAWS information is to use MP\_EXPERIMENTAL option defined in [I-D.ietf-mptcp-rfc6824bis] during initial connection setup. Or, using 'B' bit in MP\_CAPABLE option and extend MP\_CAPABLE option to convey the info. Another requirement for disabling PAWS in an MPTCP session is that DSS mapping should be put in all data segments in the session. In case an MPTCP session falls back to TCP during SYN negotiation or either side does not support disabling PAWS, PAWS MUST not be disabled if the use of TS option is negotiated.

### 3.3. TLS

When TLS [RFC5246] is used for a TCP connection, old duplicated segments can be identified as segments in the connection are protected by encryption algorithms. One difficulty for using TLS as a replacement of PAWS is that it will be hard for TCP layer to know whether TLS is used or not. One possible way is to presume the use of TLS from port numbers (e.g. 443). Or, providing APIs to signal TCP layer can be another way although this will require to update applications. In addition, TCP needs to check if the other end supports disabling PAWS. In order for this, we will need to define a new TCP option in order to be used during SYN exchange. Another possibility is to utilize timestamp values in SYN segments in order to encode the feature negotiation information. An example of using timestamp value in SYN segments for feature negotiation is described in [I-D.scheffenegger-tcpm-timestamp-negotiation]. When either side does not support disabling PAWS, PAWS MUST not be disabled if the use of TS option is negotiated.

#### 4. Duplicates from Earlier Connection Incarnations

As described in [RFC7323], the main purpose of PAWS is to protect against old duplicates from the same connection. In addition, it is expected to provide additional security against old duplicates from earlier connections. Although this feature is not strongly encouraged in the RFC, some implementations support it. We believe the protection described above can be used for this purpose as well. By using encryption logics or extended sequence number space, they can distinguish between the packets from the current connection and the packets from earlier connections. The protections will even be stronger than the protection PAWS can provide. We believe the replacements of PAWS will also be able to facilitate recycling the connections in TIME\_WAIT. For example, it was reported some implementations replaced the connections in TIME\_WAIT by a new incoming connection with the same 4 tuples when its timestamp is newer than the one in TIME\_WAIT. However, this logic will not work properly when multiple clients behind NAT or when a node doesn't maintain global timestamp offset across all TCP connections. On the other hand, when a TCP connection can utilize the protections described above, it can recycle the connection in TIME\_WAIT with more robust algorithms.

#### 5. Security Considerations

TBD

#### 6. IANA Considerations

This document may use the Experimental Option Experiment Identifier defined in [RFC6994]. In this case, an application for this codepoint in the IANA TCP Experimental Option ExID registry will be submitted.

#### 7. References

##### 7.1. Normative References

[I-D.ietf-mptcp-rfc6824bis]

Ford, A., Raiciu, C., Handley, M., Bonaventure, O., and C. Paasch, "TCP Extensions for Multipath Operation with Multiple Addresses", draft-ietf-mptcp-rfc6824bis-11 (work in progress), May 2018.

[I-D.ietf-tcpinc-tcpeno]

Bittau, A., Giffin, D., Handley, M., Mazieres, D., and E. Smith, "TCP-ENO: Encryption Negotiation Option", draft-ietf-tcpinc-tcpeno-18 (work in progress), 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>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<https://www.rfc-editor.org/info/rfc5246>>.
- [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>>.
- [RFC7323] Borman, D., Braden, B., Jacobson, V., and R. Scheffenegger, Ed., "TCP Extensions for High Performance", RFC 7323, DOI 10.17487/RFC7323, September 2014, <<https://www.rfc-editor.org/info/rfc7323>>.

## 7.2. Informative References

- [I-D.scheffenegger-tcpm-timestamp-negotiation] Scheffenegger, R., Kuehlewind, M., and B. Trammell, "Additional negotiation in the TCP Timestamp Option field during the TCP handshake", draft-scheffenegger-tcpm-timestamp-negotiation-05 (work in progress), October 2012.
- [MALLMAN99] Allman, M. and V. Paxson, "On Estimating End-to-End Network Path Properties", Proceedings of the ACM SIGCOMM , September 1999.
- [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>>.
- [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>>.
- [RFC6994] Touch, J., "Shared Use of Experimental TCP Options", RFC 6994, DOI 10.17487/RFC6994, August 2013, <<https://www.rfc-editor.org/info/rfc6994>>.
- [TCPINC] The IETF, "The TCPINC Working Group, <https://datatracker.ietf.org/wg/tcpinc/documents/>", 2014.

Author's Address

Yoshifumi Nishida  
GE Global Research  
2623 Camino Ramon  
San Ramon, CA 94583  
USA

Email: nishida@wide.ad.jp