TCP Maintenance Working Group
Internet-Draft
Intended status: Experimental
Expires: January 7, 2017

Y. Cheng
N. Cardwell
Google, Inc

July 6, 2016

RACK: a time-based fast loss detection algorithm for TCP
draft-cheng-tcpm-rack-01

Abstract

This document presents a new TCP loss detection algorithm called RACK ("Recent ACKnowledgment"). RACK uses the notion of time, instead of packet or sequence counts, to detect losses, for modern TCP implementations that can support per-packet timestamps and the selective acknowledgment (SACK) option. It is intended to replace the conventional DUPACK threshold approach and its variants, as well as other nonstandard approaches.

Status of This Memo

Copyright Notice

1. Introduction

This document presents a new loss detection algorithm called RACK ("Recent ACKnowledgment"). RACK uses the notion of time instead of the conventional packet or sequence counting approaches for detecting losses. RACK deems a packet lost if some packet sent sufficiently later has been delivered. It does this by recording packet transmission times and inferring losses using cumulative acknowledgments or selective acknowledgment (SACK) TCP options.

In the last couple of years we have been observing several increasingly common loss and reordering patterns in the Internet:

1. Lost retransmissions. Traffic policers [POLICER16] and burst losses often cause retransmissions to be lost again, severely increasing TCP latency.

2. Tail drops. Structured request-response traffic turns more losses into tail drops. In such cases, TCP is application-limited, so it cannot send new data to probe losses and has to rely on retransmission timeouts (RTOs).

3. Reordering. Link layer protocols (e.g., 802.11 block ACK) or routers' internal load-balancing can deliver TCP packets out of order. The degree of such reordering is usually within the order of the path round trip time.

Despite TCP stacks (e.g. Linux) that implement many of the standard and proposed loss detection algorithms [RFC3517][RFC4653][RFC5827][RFC5681][RFC6675][RFC7765][FACK][THIN-STREAM][TLP], we've found that together they do not perform well. The main reason is that many of them are based on the classic rule of counting duplicate acknowledgments [RFC5681]. They can either detect loss quickly or accurately, but not both, especially when the sender is application-limited or under reordering that is unpredictable. And under these conditions none of them can detect lost retransmissions well.

Also, these algorithms, including RFCs, rarely address the interactions with other algorithms. For example, FACK may consider a packet is lost while RFC3517 may not. Implementing N algorithms while dealing with N^2 interactions is a daunting task and error-prone.

The goal of RACK is to solve all the problems above by replacing many of the loss detection algorithms above with one simpler, and also more effective, algorithm.

2. Overview

The main idea behind RACK is that if a packet has been delivered out of order, then the packets sent chronologically before that were either lost or reordered. This concept is not fundamentally different from [RFC5681][RFC3517][FACK]. But the key innovation in RACK is to use a per-packet transmission timestamp and widely deployed SACK options to conduct time-based inferences instead of inferring losses with packet or sequence counting approaches.

Using a threshold for counting duplicate acknowledgments (i.e., dupthresh) is no longer reliable because of today’s prevalent reordering patterns. A common type of reordering is that the last "runt" packet of a window’s worth of packet bursts gets delivered first, then the rest arrive shortly after in order. To handle this effectively, a sender would need to constantly adjust the dupthresh to the burst size; but this would risk increasing the frequency of RTOs on real losses.

Today’s prevalent lost retransmissions also cause problems with packet-counting approaches [RFC5681][RFC3517][FACK], since those approaches depend on reasoning in sequence number space. Retransmissions break the direct correspondence between ordering in sequence space and ordering in time. So when retransmissions are lost, sequence-based approaches are often unable to infer and quickly repair losses that can be deduced with time-based approaches.

Instead of counting packets, RACK uses the most recently delivered packet’s transmission time to judge if some packets sent previous to that time have "expired" by passing a certain reordering settling window. On each ACK, RACK marks any already-expired packets lost, and for any packets that have not yet expired it waits until the reordering window passes and then marks those lost as well. In either case, RACK can repair the loss without waiting for a (long) RTO. RACK can be applied to both fast recovery and timeout recovery, and can detect losses on both originally transmitted and retransmitted packets, making it a great all-weather recovery mechanism.

3. Requirements

The reader is expected to be familiar with the definitions given in the TCP congestion control [RFC5681] and selective acknowledgment

[RFC2018] RFCs. Familiarity with the conservative SACK-based recovery for TCP [RFC6675] is not expected but helps.

RACK has three requirements:

1. The connection MUST use selective acknowledgment (SACK) options [RFC2018].

2. For each packet sent, the sender MUST store its most recent transmission time with (at least) millisecond granularity. For round-trip times lower than a millisecond (e.g., intra-datacenter communications) microsecond granularity would significantly help the detection latency but is not required.

3. For each packet sent, the sender MUST store 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. For the ease of explaining the algorithm, we use a pseudo-scoreboard that manages the data in sequence number ranges. But the specifics of the data structure are left to the implementor.

RACK does not need any change on the receiver.

4. Definitions of variables

A sender needs to store these new RACK variables:

"Packet.xmit_ts" is the time of the last transmission of a data packet, including any 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.xmit_ts" is the most recent Packet.xmit_ts among all the packets that were delivered (either cumulatively acknowledged or selectively acknowledged) on the connection.

"RACK.end_seq" is the ending TCP sequence number of the packet that was used to record the RACK.xmit_ts above.

"RACK.RTT" is the associated RTT measured when RACK.xmit_ts, above, was changed. It is the RTT of the most recently transmitted packet that has been delivered (either cumulatively acknowledged or selectively acknowledged) on the connection.

"RACK.reo_wnd" is a reordering window for the connection, computed in the unit of time used for recording packet transmission times. It is used to defer the moment at which RACK marks a packet lost.

"RACK.min_RTT" is the estimated minimum round-trip time (RTT) of the connection.

Note that the Packet.xmit_ts variable is per packet in flight. The RACK.xmit_ts, RACK.RTT, RACK.reo_wnd, and RACK.min_RTT variables are per connection.

5. Algorithm Details

5.1. Transmitting a data packet

Upon transmitting a new packet or retransmitting an old packet, record the time in Packet.xmit_ts. RACK does not care if the retransmission is triggered by an ACK, new application data, an RTO, or any other means.

5.2. Upon receiving an ACK

Step 1: Update RACK.min_RTT.

Use the RTT measurements obtained in [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.reo_wnd.

To handle the prevalent small degree of reordering, RACK.reo_wnd serves as an allowance for settling time before marking a packet lost. By default it is 1 millisecond. We RECOMMEND implementing the reordering detection in [REORDER-DETECT][RFC4737] to dynamically adjust the reordering window. When the sender detects packet reordering RACK.reo_wnd MAY be changed to RACK.min_RTT/4. We discuss more about the reordering window in the next section.

Step 3: Advance RACK.xmit_ts and update RACK.RTT and RACK.end_seq

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. Ignore the packet if any of its TCP sequences has been retransmitted before and either of two condition is true:

1. The Timestamp Echo Reply field (TSecr) of the ACK's timestamp option [RFC7323], if available, indicates the ACK was not acknowledging the last retransmission of the packet.

2. The packet was last retransmitted less than RACK.min_rtt ago. While it is still possible the packet is spuriously retransmitted because of a recent RTT decrease, we believe that our experience suggests this is a reasonable heuristic.

If this ACK causes a change to RACK.xmit_ts then record the RTT and sequence implied by this ACK:

```
RACK.RTT = Now() - RACK.xmit_ts
RACK.end_seq = Packet.end_seq
```

Exit here and omit the following steps if RACK.xmit_ts has not changed.

Step 4: Detect losses.

For each packet that has not been fully 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, the packet was likely lost.

If a packet that was sent later has been delivered, but the reordering window has not passed, then it is not yet safe to deem the given 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 the S/ACK of the RACK packet (RACK.ack_ts) and now

So we mark a packet as lost if:

```
RACK.xmit_ts > Packet.xmit_ts   AND
(RACK.xmit_ts - Packet.xmit_ts) + (now - RACK.ack_ts) > RACK.reo_wnd
```

If we solve this second condition for "now", the moment at which we can declare a packet lost, then we get:

```
now > Packet.xmit_ts + RACK.reo_wnd + (RACK.ack_ts - RACK.xmit_ts)
```

Then (RACK.ack_ts - RACK.xmit_ts) is just the RTT of the packet we used to set RACK.xmit_ts, so this reduces to:

```
now > Packet.xmit_ts + RACK.RTT + RACK.reo_wnd
```

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():
 min_timeout = 0

 For each packet, Packet, in the scoreboard:
     If Packet is already SACKed, ACKed,
        or marked lost and not yet retransmitted:
         Skip to the next packet

     If Packet.xmit_ts > RACK.xmit_ts:
         Skip to the next packet
     If Packet.xmit_ts == RACK.xmit_ts AND // Timestamp tie breaker
Packet.end_seq > RACK.end_seq
         Skip to the next packet

     timeout = Packet.xmit_ts + RACK.RTT + RACK.reo_wnd + 1
     If Now() >= timeout
         Mark Packet lost
     Else If (min_timeout == 0) or (timeout is before min_timeout):
         min_timeout = timeout

 If min_timeout != 0
     Arm a timer to call RACK_detect_loss() after min_timeout
```

6. Analysis and Discussion

6.1. Advantages

The biggest advantage of RACK is that every data packet, whether it is an original data transmission or a retransmission, can be used to detect losses of the packets sent prior to it.

Example: tail drop. Consider a sender that transmits a window of three data packets (P1, P2, P3), and P1 and P3 are lost. Suppose the transmission of each packet is at least RACK.reo_wnd (1 millisecond by default) after the transmission of the previous packet. RACK will mark P1 as lost when the SACK of P2 is received, and this will trigger the retransmission of P1 as R1. When R1 is cumulatively acknowledged, RACK will mark P3 as lost and the sender will retransmit P3 as R3. This example illustrates how RACK is able to repair certain drops at the tail of a transaction without any timer. Notice that neither the conventional duplicate ACK threshold [RFC5681], nor [RFC6675], nor the Forward Acknowledgment [FACK] algorithm can detect such losses, because of the required packet or sequence count.

Example: lost retransmit. Consider a window of three data packets (P1, P2, P3) that are sent; P1 and P2 are dropped. Suppose the transmission of each packet is at least RACK.reo_wnd (1 millisecond by default) after the transmission of the previous packet. When P3 is SACKed, RACK will mark P1 and P2 lost and they will be retransmitted as R1 and R2. Suppose R1 is lost again (as a tail drop) 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 due to application-limited behavior. Suppose the sender has detected reordering previously (e.g., by implementing the algorithm in [REORDER-DETECT]) and thus RACK.reo_wnd is min_RTT/4. Now P3 is reordered and delivered first, before P1 and P2. As long as P1 and P2 are delivered within min_RTT/4, RACK will not consider P1 and P2 lost. But if P1 and P2 are delivered outside the reordering window, then RACK will still falsely mark P1 and P2 lost. We discuss how to reduce the false positives in the end of this section.

The examples above show that RACK is particularly useful when the sender is limited by the application, which is common for interactive, request/response traffic. Similarly, RACK still works when the sender is limited by the receive window, which is common for applications that use the receive window to throttle the sender.

For some implementations (e.g., Linux), RACK works quite efficiently with TCP Segmentation Offload (TSO). RACK always marks the entire TSO blob lost because the packets in the same TSO blob have the same transmission timestamp. By contrast, the counting based algorithms (e.g., [RFC3517][RFC5681]) may mark only a subset of packets in the TSO blob lost, forcing the stack to perform expensive fragmentation of the TSO blob, or to selectively tag individual packets lost in the scoreboard.

6.2. Disadvantages

RACK requires the sender to record the transmission time of each packet sent at a clock granularity of one millisecond or finer. TCP implementations that record this already for RTT estimation do not require any new per-packet state. But implementations that are not yet recording packet transmission times will need to add per-packet internal state (commonly either 4 or 8 octets per packet) to track transmission times. In contrast, the conventional approach requires one variable to track number of duplicate ACK threshold.

6.3. Adjusting the reordering window

RACK uses a reordering window of min_rtt / 4. It uses the minimum RTT to accommodate reordering introduced by packets traversing slightly different paths (e.g., router-based parallelism schemes) or out-of-order deliveries in the lower link layer (e.g., wireless links using link-layer retransmission). Alternatively, RACK can use the smoothed RTT used in RTT estimation [RFC6298]. However, smoothed RTT can be significantly inflated by orders of magnitude due to congestion and buffer-bloat, which would result in an overly conservative reordering window and slow loss detection. Furthermore, RACK uses a quarter of minimum RTT because Linux TCP uses 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.

One potential improvement is to further adapt the reordering window by measuring the degree of reordering in time, instead of packet distances. But that requires storing the delivery timestamp of each packet. Some scoreboard implementations currently merge SACKed packets together to support TSO (TCP Segmentation Offload) for faster scoreboard indexing. Supporting per-packet delivery timestamps is difficult in such implementations. However, we acknowledge that the current metric can be improved by further research.

6.4. Relationships with other loss recovery algorithms

The primary motivation of RACK is to ultimately provide a simple and general replacement for some of the standard loss recovery algorithms [RFC5681][RFC6675][RFC5827][RFC4653] and nonstandard ones [FACK][THIN-STREAM]. While RACK can be a supplemental loss detection on top of these algorithms, this is not necessary, because the 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, we RECOMMEND TCP implementations use both RACK and the algorithm specified in Section 3.2 in [RFC5681] 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 timer calculation nor detects spurious timeouts.

Furthermore, RACK naturally works well with Tail Loss Probe [TLP] because a tail loss probe solicit seither 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 can be modified to retransmit the first unacknowledged packet, which can improve application latency.

6.5. Interaction with congestion control

RACK intentionally decouples loss detection from congestion control. RACK only detects losses; it does not modify the congestion control algorithm [RFC5681][RFC6937]. However, RACK may detect losses earlier or later than the conventional duplicate ACK threshold approach does. A packet marked lost by RACK SHOULD NOT be retransmitted until congestion control deems this appropriate (e.g. using [RFC6937]).

RACK is applicable for both fast recovery and recovery after a retransmission timeout (RTO) in [RFC5681]. The distinction between fast recovery or RTO recovery is not necessary 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.

6.6. RACK for other transport protocols

RACK can be implemented in other transport protocols. The algorithm can skip step 3 and simplify if the protocol can support unique transmission or packet identifier (e.g. TCP echo options). For example, the QUIC protocol implements RACK [QUIC-LR] .

7. 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.

8. IANA Considerations

This document makes no request of IANA.

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

9. Acknowledgments

The authors thank Matt Mathis for his insights in FACK and Michael Welzl for his per-packet timer idea that inspired this work. Nandita Dukkipati, Eric Dumazet, Randy Stewart, Van Jacobson, Ian Swett, and Jana Iyengar contributed to the algorithm and the implementations in Linux, FreeBSD and QUIC.

10. References

10.1. Normative References

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

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

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

[RFC4737] Morton, A., Ciavattone, L., Ramachandran, G., Shalunov, S., and J. Perser, "Packet Reordering Metrics", RFC 4737, November 2006.

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

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

[RFC5827] Allman, M., Ayesta, U., Wang, L., Blanton, J., and P. Hurtig, "Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)", RFC 5827, April 2010.

[RFC5682] Sarolahti, P., Kojo, M., Yamamoto, K., and M. Hata, "Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP", RFC 5682, September 2009.

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

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

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

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

10.2. 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.

[TLP] Dukkipati, N., Cardwell, N., Cheng, Y., and M. Mathis, "Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Drops", draft-dukkipati-tcpm-tcp-loss-probe-01 (work in progress), August 2013.

[RFC7765] Hurtig, P., Brunstrom, A., Petlund, A., and M. Welzl, "TCP and SCTP RTO Restart", February 2016.

[REORDER-DETECT] Zimmermann, A., Schulte, L., Wolff, C., and A. Hannemann, "Detection and Quantification of Packet Reordering with TCP", draft-zimmermann-tcpm-reordering-detection-02 (work in progress), November 2014.

[QUIC-LR] Iyengar, J. and I. Swett, "QUIC Loss Recovery And Congestion Control", draft-tsvwg-quic-loss-recovery-01 (work in progress), June 2016.

[THIN-STREAM] Petlund, A., Evensen, K., Griwodz, C., and P. Halvorsen, "TCP enhancements for interactive thin-stream applications", NOSSDAV , 2008.

[SCWA99] Savage, S., Cardwell, N., Wetherall, D., and T. Anderson, "TCP Congestion Control With a Misbehaving Receiver", ACM Computer Communication Review, 29(5) , 1999.

[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.

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

Network Working Group S. Bensley
Internet-Draft D. Thaler
Intended status: Informational P. Balasubramanian
Expires: March 2, 2018 Microsoft
L. Eggert
NetApp
G. Judd
Morgan Stanley
August 29, 2017

Datacenter TCP (DCTCP): TCP Congestion Control for Datacenters
draft-ietf-tcpm-dctcp-10

Abstract

This informational memo describes Datacenter TCP (DCTCP), a TCP congestion control scheme for datacenter traffic. DCTCP extends the Explicit Congestion Notification (ECN) processing to estimate the fraction of bytes that encounter congestion, rather than simply detecting that some congestion has occurred. DCTCP then scales the TCP congestion window based on this estimate. This method achieves high burst tolerance, low latency, and high throughput with shallow-buffered switches. This memo also discusses deployment issues related to the coexistence of DCTCP and conventional TCP, the lack of a negotiating mechanism between sender and receiver, and presents some possible mitigations. This memo documents DCTCP as currently implemented by several major operating systems. DCTCP as described in this draft is applicable to deployments in controlled environments like datacenters but it must not be deployed over the public Internet without additional measures.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Large datacenters necessarily need many network switches to interconnect their many servers. Therefore, a datacenter can greatly reduce its capital expenditure by leveraging low-cost switches. However, such low-cost switches tend to have limited queue capacities and are thus more susceptible to packet loss due to congestion.

Network traffic in a datacenter is often a mix of short and long flows, where the short flows require low latencies and the long flows require high throughputs. Datacenters also experience incast bursts, where many servers send traffic to a single server at the same time. For example, this traffic pattern is a natural consequence of MapReduce [MAPREDUCE] workload: The worker nodes complete at approximately the same time, and all reply to the master node concurrently.

These factors place some conflicting demands on the queue occupancy of a switch:

- o The queue must be short enough that it does not impose excessive latency on short flows.
- o The queue must be long enough to buffer sufficient data for the long flows to saturate the path capacity.
- o The queue must be long enough to absorb incast bursts without excessive packet loss.

Standard TCP congestion control [RFC5681] relies on packet loss to detect congestion. This does not meet the demands described above. First, short flows will start to experience unacceptable latencies before packet loss occurs. Second, by the time TCP congestion control kicks in on the senders, most of the incast burst has already been dropped.

[RFC3168] describes a mechanism for using Explicit Congestion Notification (ECN) from the switches for detection of congestion. However, this method only detects the presence of congestion, not its extent. In the presence of mild congestion, the TCP congestion window is reduced too aggressively and this unnecessarily reduces the throughput of long flows.

Datacenter TCP (DCTCP) changes traditional ECN processing by estimating the fraction of bytes that encounter congestion, rather than simply detecting that some congestion has occurred. DCTCP then scales the TCP congestion window based on this estimate. This method achieves high burst tolerance, low latency, and high throughput with shallow-buffered switches. DCTCP is a modification to the processing of ECN by a conventional TCP and requires that standard TCP congestion control be used for handling packet loss.

DCTCP should only be deployed in an intra-datacenter environment where both endpoints and the switching fabric are under a single administrative domain. DCTCP MUST NOT be deployed over the public Internet without additional measures, as detailed in Section 5.

The objective of this Informational RFC is to document DCTCP as a new approach to address TCP congestion control in data centers that is known to be widely implemented and deployed. It is consensus in the IETF TCPM working group that a DCTCP standard would require further work. A precise documentation of running code enables follow-up IETF Experimental or Standards Track RFCs.

This document describes DCTCP as implemented in Microsoft Windows Server 2012 [WINDOWS]. The Linux [LINUX] and FreeBSD [FREEBSD] operating systems have also implemented support for DCTCP in a way that is believed to follow this document. Deployment experiences with DCTCP as have been documented in [MORGANSTANLEY].

2. Terminology

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

Normative language is used to describe how necessary the various aspects of a DCTCP implementation are for interoperability, but even compliant implementations without the measures in sections 4-6 would still only be safe to deploy in controlled environments, i.e., not over the public Internet.

3. DCTCP Algorithm

There are three components involved in the DCTCP algorithm:

- o The switches (or other intermediate devices in the network) detect congestion and set the Congestion Encountered (CE) codepoint in the IP header.
- o The receiver echoes the congestion information back to the sender, using the ECN-Echo (ECE) flag in the TCP header.
- o The sender computes a congestion estimate and reacts, by reducing the TCP congestion window accordingly (cwnd).

3.1. Marking Congestion on the L3 Switches and Routers

The level-3 (L3) switches and routers in a datacenter fabric indicate congestion to the end nodes by setting the CE codepoint in the IP header as specified in Section 5 of [RFC3168]. For example, the switches may be configured with a congestion threshold. When a packet arrives at a switch and its queue length is greater than the congestion threshold, the switch sets the CE codepoint in the packet. For example, Section 3.4 of [DCTCP10] suggests threshold marking with a threshold K > (RTT * C)/7, where C is the link rate in packets per second. In typical deployments the marking threshold is set to be a small value to maintain a short average queueing delay. However, the actual algorithm for marking congestion is an implementation detail of the switch and will generally not be known to the sender and receiver. Therefore, sender and receiver should not assume that a particular marking algorithm is implemented by the switching fabric.

3.2. Echoing Congestion Information on the Receiver

According to Section 6.1.3 of [RFC3168], the receiver sets the ECE flag if any of the packets being acknowledged had the CE code point set. The receiver then continues to set the ECE flag until it receives a packet with the Congestion Window Reduced (CWR) flag set. However, the DCTCP algorithm requires more detailed congestion information. In particular, the sender must be able to determine the number of bytes sent that encountered congestion. Thus, the scheme described in [RFC3168] does not suffice.

One possible solution is to ACK every packet and set the ECE flag in the ACK if and only if the CE code point was set in the packet being acknowledged. However, this prevents the use of delayed ACKs, which are an important performance optimization in datacenters. If the delayed ACK frequency is m, then an ACK is generated every m packets. The typical value of m is 2 but it could be affected by ACK throttling or packet coalescing techniques designed to improve performance.

Instead, DCTCP introduces a new Boolean TCP state variable, "DCTCP Congestion Encountered" (DCTCP.CE), which is initialized to false and stored in the Transmission Control Block (TCB). When sending an ACK, the ECE flag MUST be set if and only if DCTCP.CE is true. When receiving packets, the CE codepoint MUST be processed as follows:

1. If the CE codepoint is set and DCTCP.CE is false, set DCTCP.CE to true and send an immediate ACK.

2. If the CE codepoint is not set and DCTCP.CE is true, set DCTCP.CE to false and send an immediate ACK.

3. Otherwise, ignore the CE codepoint.

Since the immediate ACK reflects the new DCTCP.CE state, it may acknowledge any previously unacknowledged packets in the old state. This can lead to an incorrect rate computation at the sender per Section 3.3. To avoid this, an implementation MAY choose to send two ACKs, one for previously unacknowledged packets and another acknowledging the most recently received packet.

Receiver handling of the "Congestion Window Reduced" (CWR) bit is also per [RFC3168] including [RFC3168-ERRATA3639]. That is, on receipt of a segment with both the CE and CWR bits set, CWR is processed first and then CE is processed.

```
                           Send immediate
                           ACK with ECE=0
               .-----.     .--------------.     .-----.
  Send 1 ACK  /      v     v              |     |      \
   for every |     .------------.    .------------.     | Send 1 ACK
   m packets |     | DCTCP.CE=0 |    | DCTCP.CE=1 |     | for every
  with ECE=0 |     '------------'    '------------'     | m packets
              \      |     |              ^     ^      /  with ECE=1
               '-----'     '--------------'     '-----'
                            Send immediate
                            ACK with ECE=1
```

Figure 1: ACK generation state machine. DCTCP.CE abbreviated as CE.

3.3. Processing Echoed Congestion Indications on the Sender

The sender estimates the fraction of bytes sent that encountered congestion. The current estimate is stored in a new TCP state variable, DCTCP.Alpha, which is initialized to 1 and SHOULD be updated as follows:

DCTCP.Alpha = DCTCP.Alpha * (1 - g) + g * M

where

o g is the estimation gain, a real number between 0 and 1. The selection of g is left to the implementation. See Section 4 for further considerations.

o M is the fraction of bytes sent that encountered congestion during the previous observation window, where the observation window is chosen to be approximately the Round Trip Time (RTT). In particular, an observation window ends when all bytes in flight at the beginning of the window have been acknowledged.

In order to update DCTCP.Alpha, the TCP state variables defined in [RFC0793] are used, and three additional TCP state variables are introduced:

o DCTCP.WindowEnd: The TCP sequence number threshold when one observation window ends and another is to begin; initialized to SND.UNA.

- o DCTCP.BytesAcked: The number of sent bytes acknowledged during the current observation window; initialized to zero.

- o DCTCP.BytesMarked: The number of bytes sent during the current observation window that encountered congestion; initialized to zero.

The congestion estimator on the sender MUST process acceptable ACKs as follows:

1. Compute the bytes acknowledged (TCP SACK options [RFC2018] are ignored for this computation):

   BytesAcked = SEG.ACK - SND.UNA

2. Update the bytes sent:

   DCTCP.BytesAcked += BytesAcked

3. If the ECE flag is set, update the bytes marked:

   DCTCP.BytesMarked += BytesAcked

4. If the acknowledgment number is less than or equal to DCTCP.WindowEnd, stop processing. Otherwise, the end of the observation window has been reached, so proceed to update the congestion estimate as follows:

5. Compute the congestion level for the current observation window:

   M = DCTCP.BytesMarked / DCTCP.BytesAcked

6. Update the congestion estimate:

   DCTCP.Alpha = DCTCP.Alpha * (1 - g) + g * M

7. Determine the end of the next observation window:

   DCTCP.WindowEnd = SND.NXT

8. Reset the byte counters:

   DCTCP.BytesAcked = DCTCP.BytesMarked = 0

9. Rather than always halving the congestion window as described in [RFC3168], the sender SHOULD update cwnd as follows:

   cwnd = cwnd * (1 - DCTCP.Alpha / 2)

Just as specified in [RFC3168], DCTCP does not react to congestion indications more than once for every window of data. The setting of the "Congestion Window Reduced" (CWR) bit is also as per [RFC3168]. This is required for interop with classic ECN receivers due to potential misconfigurations.

3.4. Handling of Congestion Window Growth

A DCTCP sender grows its congestion window in the same way as conventional TCP. Slow start and congestion avoidance algorithms are handled as specified in [RFC5681].

3.5. Handling of Packet Loss

A DCTCP sender MUST react to loss episodes in the same way as conventional TCP, including fast retransmit and fast recovery algorithms, as specified in [RFC5681]. For cases where the packet loss is inferred and not explicitly signaled by ECN, the cwnd and other state variables like ssthresh MUST be changed in the same way that a conventional TCP would have changed them. As with ECN, DCTCP sender will only reduce the cwnd once per window of data across all loss signals. Just as specified in [RFC5681], upon a timeout, the cwnd MUST be set to no more than the loss window (1 full-sized segment), regardless of previous cwnd reductions in a given window of data.

3.6. Handling of SYN, SYN-ACK, RST Packets

If SYN, SYN-ACK and RST packets for DCTCP connections have the "ECN Capable Transport" (ECT) codepoint set in the IP header, they will receive the same treatment as other DCTCP packets when forwarded by a switching fabric under load. Lack of ECT in these packets can result in a higher drop rate depending on the switching fabric configuration. Hence for DCTCP connections, the sender SHOULD set ECT for SYN, SYN-ACK and RST packets. A DCTCP receiver ignores CE codepoints set on any SYN, SYN-ACK, or RST packets.

4. Implementation Issues

4.1. Configuration of DCTCP

An implementation needs to know when to use DCTCP. Datacenter servers may need to communicate with endpoints outside the datacenter, where DCTCP is unsuitable or unsupported. Thus, a global configuration setting to enable DCTCP will generally not suffice. DCTCP provides no mechanism for negotiating its use. Thus, additional management and configuration functionality is needed to ensure that DCTCP is not used with non-DCTCP endpoints.

Known solutions rely on either configuration or heuristics. Heuristics need to allow endpoints to individually enable DCTCP, to ensure a DCTCP sender is always paired with a DCTCP receiver. One approach is to enable DCTCP based on the IP address of the remote endpoint. Another approach is to detect connections that transmit within the bounds a datacenter. For example, an implementation could support automatic selection of DCTCP if the estimated RTT is less than a threshold (like 10 msec) and ECN is successfully negotiated, under the assumption that if the RTT is low, then the two endpoints are likely in the same datacenter network.

[RFC3168] forbids the ECN-marking of pure ACK packets, because of the inability of TCP to mitigate ACK-path congestion. RFC 3168 also forbids ECN-marking of retransmissions, window probes and RSTs. However, dropping all these control packets - rather than ECN marking them - has considerable performance disadvantages. It is RECOMMENDED that an implementation provide a configuration knob that will cause ECT to be set on such control packets, which can be used in environments where such concerns do not apply. See [ECN-EXPERIMENTATION] for details.

It is useful to implement DCTCP as additional actions on top of an existing congestion control algorithm like Reno [RFC5681]. The DCTCP implementation MAY also allow configuration of resetting the value of DCTCP.Alpha as part of processing any loss episodes.

4.2. Computation of DCTCP.Alpha

As noted in Section 3.3, the implementation will need to choose a suitable estimation gain. [DCTCP10] provides a theoretical basis for selecting the gain. However, it may be more practical to use experimentation to select a suitable gain for a particular network and workload. A fixed estimation gain of 1/16 is used in some implementations. (It should be noted that values of 0 or 1 for g result in problematic behavior; g=0 fixes DCTCP.Alpha to its initial value and g=1 sets it to M without any smoothing.)

The DCTCP.Alpha computation as per the formula in Section 3.3 involves fractions. An efficient kernel implementation MAY scale the DCTCP.Alpha value for efficient computation using shift operations. For example, if the implementation chooses g as 1/16, multiplications of DCTCP.Alpha by g become right-shifts by 4. A scaling implementation SHOULD ensure that DCTCP.Alpha is able to reach zero once it falls below the smallest shifted value (16 in the above example). At the other extreme, a scaled update needs to ensure DCTCP.Alpha does not exceed the scaling factor, which would be equivalent to greater than 100% congestion. So, DCTCP.Alpha MUST be clamped after an update.

This results in the following computations replacing steps 5 and 6 in Section 3.3, where SCF is the chosen scaling factor (65536 in the example) and SHF is the shift factor (4 in the example):

1. Compute the congestion level for the current observation window:

```
ScaledM = SCF * DCTCP.BytesMarked / DCTCP.BytesAcked
```

2. Update the congestion estimate:

```
if (DCTCP.Alpha >> SHF) == 0 then DCTCP.Alpha = 0

DCTCP.Alpha += (ScaledM >> SHF) - (DCTCP.Alpha >> SHF)

if DCTCP.Alpha > SCF then DCTCP.Alpha = SCF
```

5. Deployment Issues

DCTCP and conventional TCP congestion control do not coexist well in the same network. In typical DCTCP deployments, the marking threshold in the switching fabric is set to a very low value to reduce queueing delay, and a relatively small amount of congestion will exceed the marking threshold. During such periods of congestion, conventional TCP will suffer packet loss and quickly and drastically reduce cwnd. DCTCP, on the other hand, will use the fraction of marked packets to reduce cwnd more gradually. Thus, the rate reduction in DCTCP will be much slower than that of conventional TCP, and DCTCP traffic will gain a larger share of the capacity compared to conventional TCP traffic traversing the same path. If the traffic in the datacenter is a mix of conventional TCP and DCTCP, it is RECOMMENDED that DCTCP traffic be segregated from conventional TCP traffic. [MORGANSTANLEY] describes a deployment that uses the IP Differentiated Services Code Point (DSCP) bits to segregate the network such that Active Queue Management (AQM) [RFC7567] is applied to DCTCP traffic, whereas TCP traffic is managed via drop-tail queueing.

Deployments should take into account segregation of non-TCP traffic as well. Today's commodity switches allow configuration of different marking/drop profiles for non-TCP and non-IP packets. Non-TCP and non-IP packets should be able to pass through such switches, unless they really run out of buffer space.

Since DCTCP relies on congestion marking by the switches, DCTCP's potential can only be realized in datacenters where the entire network infrastructure supports ECN. The switches may also support configuration of the congestion threshold used for marking. The proposed parameterization can be configured with switches that implement Random Early Detection (RED) [RFC2309]. [DCTCP10] provides a theoretical basis for selecting the congestion threshold, but as with the estimation gain, it may be more practical to rely on experimentation or simply to use the default configuration of the device. DCTCP will revert to loss-based congestion control when packet loss is experienced (e.g. when transiting a congested drop-tail link, or a link with an AQM drop behavior).

DCTCP requires changes on both the sender and the receiver, so both endpoints must support DCTCP. Furthermore, DCTCP provides no mechanism for negotiating its use, so both endpoints must be configured through some out-of-band mechanism to use DCTCP. A variant of DCTCP that can be deployed unilaterally and only requires standard ECN behavior has been described in [ODCTCP][BSDCAN], but requires additional experimental evaluation.

6. Known Issues

DCTCP relies on the sender’s ability to reconstruct the stream of CE codepoints received by the remote endpoint. To accomplish this, DCTCP avoids using a single ACK packet to acknowledge segments received both with and without the CE codepoint set. However, if one or more ACK packets are dropped, it is possible that a subsequent ACK will cumulatively acknowledge a mix of CE and non-CE segments. This will, of course, result in a less accurate congestion estimate. There are some potential considerations:

- Even with an inaccurate congestion estimate, DCTCP may still perform better than [RFC3168].

- If the estimation gain is small relative to the packet loss rate, the estimate may not be too inaccurate.

- If ACK packet loss mostly occurs under heavy congestion, most drops will occur during an unbroken string of CE packets, and the estimate will be unaffected.

However, the effect of packet drops on DCTCP under real world conditions has not been analyzed.

DCTCP provides no mechanism for negotiating its use. The effect of using DCTCP with a standard ECN endpoint has been analyzed in [ODCTCP][BSDCAN]. Furthermore, it is possible that other implementations may also modify [RFC3168] behavior without negotiation, causing further interoperability issues.

Much like standard TCP, DCTCP is biased against flows with longer RTTs. A method for improving the RTT fairness of DCTCP has been proposed in [ADCTCP], but requires additional experimental evaluation.

7. Security Considerations

DCTCP enhances ECN and thus inherits the general security considerations discussed in [RFC3168], although additional mitigation options exist due to the limited intra-datacenter deployment of DCTCP.

The processing changes introduced by DCTCP do not exacerbate the considerations in [RFC3168] or introduce new ones. In particular, with either algorithm, the network infrastructure or the remote endpoint can falsely report congestion and thus cause the sender to reduce cwnd. However, this is no worse than what can be achieved by simply dropping packets.

[RFC3168] requires that a compliant TCP must not set ECT on SYN or SYN-ACK packets. [RFC5562] proposes setting ECT on SYN-ACK packets, but maintains the restriction of no ECT on SYN packets. Both these RFCs prohibit ECT in SYN packets due to security concerns regarding malicious SYN packets with ECT set. These RFCs, however, are intended for general Internet use, and do not directly apply to a controlled datacenter environment. The security concerns addressed by both these RFCs might not apply in controlled environments like datacenters, and it might not be necessary to account for the presence of non-ECN servers. Beyond the security considerations related to virtual servers, additional security can be imposed in the physical servers to intercept and drop traffic resembling an attack.

8. IANA Considerations

This document has no actions for IANA.

9. Acknowledgements

The DCTCP algorithm was originally proposed and analyzed in [DCTCP10] by Mohammad Alizadeh, Albert Greenberg, Dave Maltz, Jitu Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan.

We would like to thank Andrew Shewmaker for identifying the problem of clamping DCTCP.Alpha and proposing a solution for it.

Lars Eggert has received funding from the European Union’s Horizon 2020 research and innovation program 2014-2018 under grant agreement No. 644866 ("SSICLOPS"). This document reflects only the authors’ views and the European Commission is not responsible for any use that may be made of the information it contains.

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>.

[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>.

[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>.

[RFC3168-ERRATA3639] Scheffenegger, R., "RFC3168 Errata ID 3639", 2013, <http://www.rfc-editor.org/errata_search.php?rfc=3168&eid=3639>.

[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>.

[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>.

10.2. Informative References

[ADCTCP] Alizadeh, M., Javanmard, A., and B. Prabhakar, "Analysis of DCTCP: Stability, Convergence, and Fairness", DOI 10.1145/1993744.1993753, Proc. ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 11), June 2011, <https://dl.acm.org/citation.cfm?id=1993753>.

[BSDCAN] Kato, M., Eggert, L., Zimmermann, A., van Meter, R., and H. Tokuda, "Extensions to FreeBSD Datacenter TCP for Incremental Deployment Support", BSDCan 2015, June 2015, <https://www.bsdcan.org/2015/schedule/events/559.en.html>.

[DCTCP10] Alizadeh, M., Greenberg, A., Maltz, D., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and M. Sridharan, "Data Center TCP (DCTCP)", DOI 10.1145/1851182.1851192, Proc. ACM SIGCOMM 2010 Conference (SIGCOMM 10), August 2010, <http://dl.acm.org/citation.cfm?doid=1851182.1851192>.

[ECN-EXPERIMENTATION]
Black, D., "Explicit Congestion Notification (ECN) Experimentation", 2017, <https://datatracker.ietf.org/doc/draft-ietf-tsvwg-ecn-experimentation/>.

[FREEBSD] Kato, M. and H. Panchasara, "DCTCP (Data Center TCP) implementation", 2015, <https://github.com/freebsd/freebsd/commit/8ad879445281027858a7fa706d13e458095b595f>.

[LINUX] Borkmann, D. and F. Westphal, "Linux DCTCP patch", 2014, <https://git.kernel.org/cgit/linux/kernel/git/davem/net-next.git/commit/?id=e3118e8359bb7c59555aca60c725106e6d78c5ce>.

[MAPREDUCE]
Dean, J. and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", Proc. 6th ACM/USENIX Symposium on Operating Systems Design and Implementation (OSDI 04), December 2004, <https://www.usenix.org/legacy/publications/library/proceedings/osdi04/tech/dean.html>.

[MORGANSTANLEY]
Judd, G., "Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter", Proc. 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), May 2015, <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/judd>.

[ODCTCP] Kato, M., "Improving Transmission Performance with One-Sided Datacenter TCP", M.S. Thesis, Keio University, 2014, <http://eggert.org/students/kato-thesis.pdf>.

[RFC2309] Braden, B., Clark, D., Crowcroft, J., Davie, B., Deering, S., Estrin, D., Floyd, S., Jacobson, V., Minshall, G., Partridge, C., Peterson, L., Ramakrishnan, K., Shenker, S., Wroclawski, J., and L. Zhang, "Recommendations on Queue Management and Congestion Avoidance in the Internet", RFC 2309, DOI 10.17487/RFC2309, April 1998, <https://www.rfc-editor.org/info/rfc2309>.

[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>.

[WINDOWS] Microsoft, "Windows DCTCP reference", 2012, <https://technet.microsoft.com/en-us/library/hh997028(v=ws.11).aspx>.

Authors' Addresses

Stephen Bensley
Microsoft
One Microsoft Way
Redmond, WA 98052
USA

Phone: +1 425 703 5570
Email: sbens@microsoft.com

Dave Thaler
Microsoft

Phone: +1 425 703 8835
Email: dthaler@microsoft.com

Praveen Balasubramanian
Microsoft

Phone: +1 425 538 2782
Email: pravb@microsoft.com

Lars Eggert
NetApp
Sonnenallee 1
Kirchheim 85551
Germany

Phone: +49 151 120 55791
Email: lars@netapp.com
URI: http://eggert.org/

Glenn Judd
Morgan Stanley

Phone: +1 973 979 6481
Email: glenn.judd@morganstanley.com

Internet Engineering Task Force
Internet-Draft
Obsoletes: 793, 879, 2873, 6093, 6429,
6528, 6691 (if approved)
Updates: 5961, 1122 (if approved)
Intended status: Standards Track
Expires: January 31, 2020

W. Eddy, Ed.
MTI Systems

July 30, 2019

Transmission Control Protocol Specification
draft-ietf-tcpm-rfc793bis-14

Abstract

This document specifies the Internet's Transmission Control Protocol (TCP). TCP is an important transport layer protocol in the Internet stack, and has continuously evolved over decades of use and growth of the Internet. Over this time, a number of changes have been made to TCP as it was specified in RFC 793, though these have only been documented in a piecemeal fashion. This document collects and brings those changes together with the protocol specification from RFC 793. This document obsoletes RFC 793, as well as 879, 2873, 6093, 6429, 6528, and 6691 that updated parts of RFC 793. It updates RFC 1122, and should be considered as a replacement for the portions of that document dealing with TCP requirements. It updates RFC 5961 due to a small clarification in reset handling while in the SYN-RECEIVED state.

RFC EDITOR NOTE: If approved for publication as an RFC, this should be marked additionally as "STD: 7" and replace RFC 793 in that role.

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [4].

Status of This Memo

Copyright Notice

Table of Contents

1. Purpose and Scope

In 1981, RFC 793 [12] was released, documenting the Transmission Control Protocol (TCP), and replacing earlier specifications for TCP that had been published in the past.

Since then, TCP has been implemented many times, and has been used as a transport protocol for numerous applications on the Internet.

For several decades, RFC 793 plus a number of other documents have combined to serve as the specification for TCP [37]. Over time, a number of errata have been identified on RFC 793, as well as deficiencies in security, performance, and other aspects. A number of enhancements has grown and been documented separately. These were never accumulated together into an update to the base specification.

The purpose of this document is to bring together all of the IETF Standards Track changes that have been made to the basic TCP functional specification and unify them into an update of the RFC 793 protocol specification. Some companion documents are referenced for important algorithms that TCP uses (e.g. for congestion control), but have not been attempted to include in this document. This is a conscious choice, as this base specification can be used with multiple additional algorithms that are developed and incorporated separately, but all TCP implementations need to implement this specification as a common basis in order to interoperate. As some additional TCP features have become quite complicated themselves (e.g. advanced loss recovery and congestion control), future companion documents may attempt to similarly bring these together.

In addition to the protocol specification that descibes the TCP segment format, generation, and processing rules that are to be implemented in code, RFC 793 and other updates also contain informative and descriptive text for human readers to understand aspects of the protocol design and operation. This document does not attempt to alter or update this informative text, and is focused only on updating the normative protocol specification. We preserve references to the documentation containing the important explanations and rationale, where appropriate.

This document is intended to be useful both in checking existing TCP implementations for conformance, as well as in writing new implementations.

2. Introduction

RFC 793 contains a discussion of the TCP design goals and provides examples of its operation, including examples of connection establishment, closing connections, and retransmitting packets to repair losses.

This document describes the basic functionality expected in modern implementations of TCP, and replaces the protocol specification in RFC 793. It does not replicate or attempt to update the introduction and philosophy content in RFC 793 (sections 1 and 2 of that document). Other documents are referenced to provide explanation of the theory of operation, rationale, and detailed discussion of design decisions. This document only focuses on the normative behavior of the protocol.

The "TCP Roadmap" [37] provides a more extensive guide to the RFCs that define TCP and describe various important algorithms. The TCP Roadmap contains sections on strongly encouraged enhancements that improve performance and other aspects of TCP beyond the basic operation specified in this document. As one example, implementing congestion control (e.g. [25]) is a TCP requirement, but is a complex topic on its own, and not described in detail in this document, as there are many options and possibilities that do not impact basic interoperability. Similarly, most common TCP implementations today include the high-performance extensions in [35], but these are not strictly required or discussed in this document.

A list of changes from RFC 793 is contained in Section 4.

Each use of RFC 2119 keywords in the document is individually labeled and referenced in Appendix B that summarizes implementation requirements. Sentences using "MUST" are labeled as "MUST-X" with X being a numeric identifier enabling the requirement to be located easily when referenced from Appendix B. Similarly, sentences using "SHOULD" are labeled with "SHLD-X", "MAY" with "MAY-X", and "RECOMMENDED" with "REC-X". For the purposes of this labeling, "SHOULD NOT" and "MUST NOT" are labeled the same as "SHOULD" and "MUST" instances.

2.1. Key TCP Concepts

TCP provides a reliable, in-order, byte-stream service to applications.

The application byte-stream is conveyed over the network via TCP segments, with each TCP segment sent as an Internet Protocol (IP) datagram.

TCP reliability consists of detecting packet losses (via sequence numbers) and errors (via per-segment checksums), as well as correction of losses and errors via retransmission.

TCP supports unicast delivery of data. Anycast applications exist that successfully use TCP without modifications, though there is some risk of instability due to rerouting.

TCP is connection-oriented, though does not inherently include a liveness detection capability.

Data flow is supported bidirectionally over TCP connections, though applications are free to flow data only unidirectionally, if they so choose.

TCP uses port numbers to identify application services and to multiplex multiple flows between hosts.

A more detailed description of TCP's features compared to other transport protocols can be found in Section 3.1 of [40]. Further description of the motivations for developing TCP and its role in the Internet stack can be found in Section 2 of [12] and earlier versions of the TCP specification.

3. Functional Specification

3.1. Header Format

TCP segments are sent as internet datagrams. The Internet Protocol (IP) header carries several information fields, including the source and destination host addresses [1] [11]. A TCP header follows the Internet header, supplying information specific to the TCP protocol. This division allows for the existence of host level protocols other than TCP. In early development of the Internet suite of protocols, the IP header fields had been a part of TCP.

TCP Header Format

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |          Source Port          |       Destination Port        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                        Sequence Number                        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Acknowledgment Number                      |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |  Data |       |C|E|U|A|P|R|S|F|                               |
   | Offset| Rsrvd |W|C|R|C|S|S|Y|I|            Window             |
   |       |       |R|E|G|K|H|T|N|N|                               |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |           Checksum            |         Urgent Pointer        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Options                    |    Padding    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                             data                              |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Note that one tick mark represents one bit position.

Figure 1: TCP Header Format

Source Port: 16 bits

The source port number.

Destination Port: 16 bits

The destination port number.

Sequence Number: 32 bits

The sequence number of the first data octet in this segment (except when SYN is present). If SYN is present the sequence number is the initial sequence number (ISN) and the first data octet is ISN+1.

Acknowledgment Number: 32 bits

If the ACK control bit is set this field contains the value of the next sequence number the sender of the segment is expecting to receive. Once a connection is established this is always sent.

Data Offset: 4 bits

The number of 32 bit words in the TCP Header. This indicates where the data begins. The TCP header (even one including options) is an integral number of 32 bits long.

Rsrvd - Reserved: 4 bits

Reserved for future use. Must be zero in generated segments and must be ignored in received segments, if corresponding future features are unimplemented by the sending or receiving host.

Control Bits: 8 bits (from left to right):

CWR: Congestion Window Reduced (see [8])
ECE: ECN-Echo (see [8])
URG: Urgent Pointer field significant
ACK: Acknowledgment field significant
PSH: Push Function (see the Send Call description in Section 3.9.1)
RST: Reset the connection
SYN: Synchronize sequence numbers
FIN: No more data from sender

The control bits are also know as "flags". Assignment is managed by IANA from the "TCP Header Flags" registry [42].

Window: 16 bits

The number of data octets beginning with the one indicated in the acknowledgment field which the sender of this segment is willing to accept.

The window size MUST be treated as an unsigned number, or else large window sizes will appear like negative windows and TCP will now work (MUST-1). It is RECOMMENDED that implementations will reserve 32-bit fields for the send and receive window sizes in the connection record and do all window computations with 32 bits (REC-1).

Checksum: 16 bits

The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header and text. If a segment contains an odd number of header and text octets to be checksummed, the last octet is padded on the right with zeros to form a 16 bit word for checksum purposes. The pad is not transmitted as part of the segment. While computing the checksum, the checksum field itself is replaced with zeros.

The checksum also covers a pseudo header conceptually prefixed to the TCP header. The pseudo header is 96 bits for IPv4 and 320 bits for IPv6. For IPv4, this pseudo header contains the Source Address, the Destination Address, the Protocol (PTCL), and TCP length. This gives the TCP protection against misrouted segments. This information is carried in IPv4 and is transferred across the TCP/Network interface in the arguments or results of calls by the TCP on the IP.

```
+--------+--------+--------+--------+
|           Source Address          |
+--------+--------+--------+--------+
|         Destination Address       |
+--------+--------+--------+--------+
|  zero  |  PTCL  |    TCP Length   |
+--------+--------+--------+--------+
```

Psuedo header components:

Source Address: the IPv4 source address in network byte order

Destination Address: the IPv4 destination address in network byte order

zero: bits set to zero

PTCL: the protocol number from the IP header

TCP Length: the TCP header length plus the data length in octets (this is not an explicitly transmitted quantity, but is computed), and it does not count the 12 octets of the pseudo header.

For IPv6, the pseudo header is contained in section 8.1 of RFC 8200 [11], and contains the IPv6 Source Address and Destination Address, an Upper Layer Packet Length (a 32-bit value otherwise equivalent to TCP Length in the IPv4 pseudo header), three bytes of zero-padding, and a Next Header value (differing from the IPv6 header value in the case of extension headers present in between IPv6 and TCP).

The TCP checksum is never optional. The sender MUST generate it (MUST-2) and the receiver MUST check it (MUST-3).

Urgent Pointer: 16 bits

This field communicates the current value of the urgent pointer as a positive offset from the sequence number in this segment. The urgent pointer points to the sequence number of the octet following the urgent data. This field is only be interpreted in segments with the URG control bit set.

Options: variable

Options may occupy space at the end of the TCP header and are a multiple of 8 bits in length. All options are included in the checksum. An option may begin on any octet boundary. There are two cases for the format of an option:

Case 1: A single octet of option-kind.

Case 2: An octet of option-kind, an octet of option-length, and the actual option-data octets.

The option-length counts the two octets of option-kind and option-length as well as the option-data octets.

Note that the list of options may be shorter than the data offset field might imply. The content of the header beyond the End-of-Option option must be header padding (i.e., zero).

The list of all currently defined options is managed by IANA [41], and each option is defined in other RFCs, as indicated there. That set includes experimental options that can be extended to support multiple concurrent uses [34].

A given TCP implementation can support any currently defined options, but the following options MUST be supported (MUST-4) (kind indicated in octal):

Kind	Length	Meaning
0	-	End of option list.
1	-	No-Operation.
2	4	Maximum Segment Size.

A TCP MUST be able to receive a TCP option in any segment (MUST-5). A TCP MUST (MUST-6) ignore without error any TCP option it does not implement, assuming that the option has a length field (all TCP options except End of option list and No-Operation have length fields). TCP MUST be prepared to handle an illegal option length (e.g., zero) without crashing; a suggested procedure is to reset the connection and log the reason (MUST-7).

Specific Option Definitions

End of Option List

```
+--------+
|00000000|
+--------+
 Kind=0
```

This option code indicates the end of the option list. This might not coincide with the end of the TCP header according to the Data Offset field. This is used at the end of all options, not the end of each option, and need only be used if the end of the options would not otherwise coincide with the end of the TCP header.

No-Operation

```
+--------+
|00000001|
+--------+
 Kind=1
```

This option code may be used between options, for example, to align the beginning of a subsequent option on a word boundary.

There is no guarantee that senders will use this option, so receivers must be prepared to process options even if they do not begin on a word boundary.

Maximum Segment Size (MSS)

```
+--------+--------+---------+--------+
|00000010|00000100|   max seg size   |
+--------+--------+---------+--------+
 Kind=2   Length=4
```

Maximum Segment Size Option Data: 16 bits

If this option is present, then it communicates the maximum receive segment size at the TCP which sends this segment. This value is limited by the IP reassembly limit. This field may be sent in the initial connection request (i.e., in segments with the SYN control bit set) and must not be sent in other segments. If this option is not used, any segment size is allowed. A more complete description of this option is in Section 3.7.1.

Padding: variable

The TCP header padding is used to ensure that the TCP header ends and data begins on a 32 bit boundary. The padding is composed of zeros.

3.2. Terminology Overview

This section includes an overview of terminology needed to understand the detailed protocol operation in the rest of the document.

3.2.1. Key Connection State Variables

Before we can discuss very much about the operation of the TCP we need to introduce some detailed terminology. The maintenance of a TCP connection requires the remembering of several variables. We conceive of these variables being stored in a connection record called a Transmission Control Block or TCB. Among the variables stored in the TCB are the local and remote socket numbers, the IP security level and compartment of the connection (see Section 3.6 and Appendix A.1), pointers to the user's send and receive buffers, pointers to the retransmit queue and to the current segment. In addition several variables relating to the send and receive sequence numbers are stored in the TCB.

Send Sequence Variables

```
SND.UNA - send unacknowledged
SND.NXT - send next
SND.WND - send window
SND.UP  - send urgent pointer
SND.WL1 - segment sequence number used for last window update
SND.WL2 - segment acknowledgment number used for last window
          update
ISS     - initial send sequence number
```

Receive Sequence Variables

```
RCV.NXT - receive next
RCV.WND - receive window
RCV.UP  - receive urgent pointer
IRS     - initial receive sequence number
```

The following diagrams may help to relate some of these variables to the sequence space.

Send Sequence Space

```
             1         2          3          4
        ----------|----------|----------|----------
               SND.UNA    SND.NXT    SND.UNA
                                    +SND.WND

  1 - old sequence numbers which have been acknowledged
  2 - sequence numbers of unacknowledged data
  3 - sequence numbers allowed for new data transmission
  4 - future sequence numbers which are not yet allowed
```

Figure 2: Send Sequence Space

The send window is the portion of the sequence space labeled 3 in Figure 2.

Receive Sequence Space

```
                       1          2          3
                   ----------|----------|----------
                          RCV.NXT    RCV.NXT
                                    +RCV.WND

        1 - old sequence numbers which have been acknowledged
        2 - sequence numbers allowed for new reception
        3 - future sequence numbers which are not yet allowed
```

Figure 3: Receive Sequence Space

The receive window is the portion of the sequence space labeled 2 in Figure 3.

There are also some variables used frequently in the discussion that take their values from the fields of the current segment.

Current Segment Variables

SEG.SEQ - segment sequence number
SEG.ACK - segment acknowledgment number
SEG.LEN - segment length
SEG.WND - segment window
SEG.UP - segment urgent pointer

3.2.2. State Machine Overview

A connection progresses through a series of states during its lifetime. The states are: LISTEN, SYN-SENT, SYN-RECEIVED, ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT, and the fictional state CLOSED. CLOSED is fictional because it represents the state when there is no TCB, and therefore, no connection. Briefly the meanings of the states are:

LISTEN - represents waiting for a connection request from any remote TCP and port.

SYN-SENT - represents waiting for a matching connection request after having sent a connection request.

SYN-RECEIVED - represents waiting for a confirming connection request acknowledgment after having both received and sent a connection request.

ESTABLISHED - represents an open connection, data received can be delivered to the user. The normal state for the data transfer phase of the connection.

FIN-WAIT-1 - represents waiting for a connection termination request from the remote TCP, or an acknowledgment of the connection termination request previously sent.

FIN-WAIT-2 - represents waiting for a connection termination request from the remote TCP.

CLOSE-WAIT - represents waiting for a connection termination request from the local user.

CLOSING - represents waiting for a connection termination request acknowledgment from the remote TCP.

LAST-ACK - represents waiting for an acknowledgment of the connection termination request previously sent to the remote TCP (this termination request sent to the remote TCP already included an acknowledgment of the termination request sent from the remote TCP).

TIME-WAIT - represents waiting for enough time to pass to be sure the remote TCP received the acknowledgment of its connection termination request.

CLOSED - represents no connection state at all.

A TCP connection progresses from one state to another in response to events. The events are the user calls, OPEN, SEND, RECEIVE, CLOSE, ABORT, and STATUS; the incoming segments, particularly those containing the SYN, ACK, RST and FIN flags; and timeouts.

The state diagram in Figure 4 illustrates only state changes, together with the causing events and resulting actions, but addresses neither error conditions nor actions which are not connected with state changes. In a later section, more detail is offered with respect to the reaction of the TCP to events. Some state names are abbreviated or hyphenated differently in the diagram from how they appear elsewhere in the document.

NOTA BENE: This diagram is only a summary and must not be taken as the total specification. Many details are not included.

```
                         +---------+ ---------\      active OPEN
                         |  CLOSED |            \    -----------
                         +---------+<---------\   \   create TCB
```

```
                              |     ^              \   \  snd SYN
                 passive OPEN |     |   CLOSE        \   \
                 ------------ |     | ----------       \   \
                  create TCB  |     | delete TCB         \   \
                              V     |                      \   \
          rcv RST (note 1)  +---------+            CLOSE    |    \
       -------------------->|  LISTEN |          ---------- |     |
      /                     +---------+          delete TCB |     |
     /           rcv SYN      |     |     SEND              |     |
    /           -----------   |     |    -------            |     V
+--------+      snd SYN,ACK  /       \   snd SYN          +--------+
|        |<-----------------           ------------------>|        |
|  SYN   |                    rcv SYN                     |  SYN   |
|  RCVD  |<-----------------------------------------------|  SENT  |
|        |                  snd SYN,ACK                   |        |
|        |------------------           -------------------|        |
+--------+   rcv ACK of SYN  \       /  rcv SYN,ACK       +--------+
   |           --------------   |     |   -----------
   |                  x         |     |     snd ACK
   |                            V     V
   |  CLOSE                   +---------+
   | -------                  |  ESTAB  |
   | snd FIN                  +---------+
   |                   CLOSE    |     |    rcv FIN
   V                  -------   |     |    -------
+---------+          snd FIN  /       \   snd ACK          +---------+
|  FIN    |<-----------------           ------------------>|  CLOSE  |
| WAIT-1  |------------------                              |   WAIT  |
+---------+          rcv FIN  \                            +---------+
  | rcv ACK of FIN   -------   |                            CLOSE  |
  | --------------   snd ACK   |                           ------- |
  V        x                   V                           snd FIN V
+---------+                  +---------+                   +---------+
|FINWAIT-2|                  | CLOSING |                   | LAST-ACK|
+---------+                  +---------+                   +---------+
  |                rcv ACK of FIN |                 rcv ACK of FIN |
  |  rcv FIN       -------------- |    Timeout=2MSL -------------- |
  |  -------              x       V    ------------        x       V
   \ snd ACK                 +---------+delete TCB         +---------+
    ------------------------>|TIME WAIT|------------------>| CLOSED  |
                             +---------+                   +---------+
```

note 1: The transition from SYN-RECEIVED to LISTEN on receiving a RST is conditional on having reached SYN-RECEIVED after a passive open.

note 2: An unshown transition exists from FIN-WAIT-1 to TIME-WAIT if a FIN is received and the local FIN is also acknowledged.

Figure 4: TCP Connection State Diagram

3.3. Sequence Numbers

A fundamental notion in the design is that every octet of data sent over a TCP connection has a sequence number. Since every octet is sequenced, each of them can be acknowledged. The acknowledgment mechanism employed is cumulative so that an acknowledgment of sequence number X indicates that all octets up to but not including X have been received. This mechanism allows for straight-forward duplicate detection in the presence of retransmission. Numbering of octets within a segment is that the first data octet immediately following the header is the lowest numbered, and the following octets are numbered consecutively.

It is essential to remember that the actual sequence number space is finite, though very large. This space ranges from 0 to 2**32 - 1. Since the space is finite, all arithmetic dealing with sequence numbers must be performed modulo 2**32. This unsigned arithmetic preserves the relationship of sequence numbers as they cycle from 2**32 - 1 to 0 again. There are some subtleties to computer modulo arithmetic, so great care should be taken in programming the comparison of such values. The symbol "=<" means "less than or equal" (modulo 2**32).

The typical kinds of sequence number comparisons which the TCP must perform include:

(a) Determining that an acknowledgment refers to some sequence number sent but not yet acknowledged.

(b) Determining that all sequence numbers occupied by a segment have been acknowledged (e.g., to remove the segment from a retransmission queue).

(c) Determining that an incoming segment contains sequence numbers which are expected (i.e., that the segment "overlaps" the receive window).

In response to sending data the TCP will receive acknowledgments. The following comparisons are needed to process the acknowledgments.

SND.UNA = oldest unacknowledged sequence number

SND.NXT = next sequence number to be sent

SEG.ACK = acknowledgment from the receiving TCP (next sequence number expected by the receiving TCP)

SEG.SEQ = first sequence number of a segment

SEG.LEN = the number of octets occupied by the data in the segment (counting SYN and FIN)

SEG.SEQ+SEG.LEN-1 = last sequence number of a segment

A new acknowledgment (called an "acceptable ack"), is one for which the inequality below holds:

SND.UNA < SEG.ACK =< SND.NXT

A segment on the retransmission queue is fully acknowledged if the sum of its sequence number and length is less or equal than the acknowledgment value in the incoming segment.

When data is received the following comparisons are needed:

RCV.NXT = next sequence number expected on an incoming segments, and is the left or lower edge of the receive window

RCV.NXT+RCV.WND-1 = last sequence number expected on an incoming segment, and is the right or upper edge of the receive window

SEG.SEQ = first sequence number occupied by the incoming segment

SEG.SEQ+SEG.LEN-1 = last sequence number occupied by the incoming segment

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

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

or

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

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

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

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

Note that when the receive window is zero no segments should be acceptable except ACK segments. Thus, it is be possible for a TCP to maintain a zero receive window while transmitting data and receiving ACKs. However, even when the receive window is zero, a TCP must process the RST and URG fields of all incoming segments.

We have taken advantage of the numbering scheme to protect certain control information as well. This is achieved by implicitly including some control flags in the sequence space so they can be retransmitted and acknowledged without confusion (i.e., one and only one copy of the control will be acted upon). Control information is not physically carried in the segment data space. Consequently, we must adopt rules for implicitly assigning sequence numbers to control. The SYN and FIN are the only controls requiring this protection, and these controls are used only at connection opening and closing. For sequence number purposes, the SYN is considered to occur before the first actual data octet of the segment in which it occurs, while the FIN is considered to occur after the last actual data octet in a segment in which it occurs. The segment length (SEG.LEN) includes both data and sequence space occupying controls. When a SYN is present then SEG.SEQ is the sequence number of the SYN.

Initial Sequence Number Selection

The protocol places no restriction on a particular connection being used over and over again. A connection is defined by a pair of sockets. New instances of a connection will be referred to as incarnations of the connection. The problem that arises from this is -- "how does the TCP identify duplicate segments from previous incarnations of the connection?" This problem becomes apparent if the connection is being opened and closed in quick succession, or if the connection breaks with loss of memory and is then reestablished.

To avoid confusion we must prevent segments from one incarnation of a connection from being used while the same sequence numbers may still be present in the network from an earlier incarnation. We want to assure this, even if a TCP crashes and loses all knowledge of the sequence numbers it has been using. When new connections are created, an initial sequence number (ISN) generator is employed which selects a new 32 bit ISN. There are security issues that result if an off-path attacker is able to predict or guess ISN values.

The recommended ISN generator is based on the combination of a (possibly fictitious) 32 bit clock whose low order bit is incremented roughly every 4 microseconds, and a pseudorandom hash function (PRF). The clock component is intended to insure that with a Maximum Segment Lifetime (MSL), generated ISNs will be unique, since it cycles approximately every 4.55 hours, which is much longer than the MSL. This recommended algorithm is further described in RFC 1948 and builds on the basic clock-driven algorithm from RFC 793.

A TCP MUST use a clock-driven selection of initial sequence numbers (MUST-8), and SHOULD generate its Initial Sequence Numbers with the expression:

ISN = M + F(localip, localport, remoteip, remoteport, secretkey)

where M is the 4 microsecond timer, and F() is a pseudorandom function (PRF) of the connection's identifying parameters ("localip, localport, remoteip, remoteport") and a secret key ("secretkey") (SHLD-1). F() MUST NOT be computable from the outside (MUST-9), or an attacker could still guess at sequence numbers from the ISN used for some other connection. The PRF could be implemented as a cryptographic has of the concatenation of the TCP connection parameters and some secret data. For discussion of the selection of a specific hash algorithm and management of the secret key data, please see Section 3 of [32].

For each connection there is a send sequence number and a receive sequence number. The initial send sequence number (ISS) is chosen by the data sending TCP, and the initial receive sequence number (IRS) is learned during the connection establishing procedure.

For a connection to be established or initialized, the two TCPs must synchronize on each other's initial sequence numbers. This is done in an exchange of connection establishing segments carrying a control bit called "SYN" (for synchronize) and the initial sequence numbers. As a shorthand, segments carrying the SYN bit are also called "SYNs". Hence, the solution requires a suitable mechanism for picking an initial sequence number and a slightly involved handshake to exchange the ISN's.

The synchronization requires each side to send its own initial sequence number and to receive a confirmation of it in acknowledgment from the other side. Each side must also receive the other side's initial sequence number and send a confirming acknowledgment.

```
1) A --> B  SYN my sequence number is X
2) A <-- B  ACK your sequence number is X
3) A <-- B  SYN my sequence number is Y
4) A --> B  ACK your sequence number is Y
```

Because steps 2 and 3 can be combined in a single message this is called the three way (or three message) handshake.

A three way handshake is necessary because sequence numbers are not tied to a global clock in the network, and TCPs may have different mechanisms for picking the ISN's. The receiver of the first SYN has no way of knowing whether the segment was an old delayed one or not, unless it remembers the last sequence number used on the connection (which is not always possible), and so it must ask the sender to verify this SYN. The three way handshake and the advantages of a clock-driven scheme are discussed in [47].

Knowing When to Keep Quiet

To be sure that a TCP does not create a segment that carries a sequence number which may be duplicated by an old segment remaining in the network, the TCP must keep quiet for an MSL before assigning any sequence numbers upon starting up or recovering from a crash in which memory of sequence numbers in use was lost. For this specification the MSL is taken to be 2 minutes. This is an engineering choice, and may be changed if experience indicates it is desirable to do so. Note that if a TCP is reinitialized in some sense, yet retains its memory of sequence numbers in use, then it need not wait at all; it must only be sure to use sequence numbers larger than those recently used.

The TCP Quiet Time Concept

This specification provides that hosts which "crash" without retaining any knowledge of the last sequence numbers transmitted on each active (i.e., not closed) connection shall delay emitting any TCP segments for at least the agreed MSL in the internet system of which the host is a part. In the paragraphs below, an explanation for this specification is given. TCP implementors may violate the "quiet time" restriction, but only at the risk of causing some old data to be accepted as new or new data rejected as old duplicated by some receivers in the internet system.

TCPs consume sequence number space each time a segment is formed and entered into the network output queue at a source host. The duplicate detection and sequencing algorithm in the TCP protocol relies on the unique binding of segment data to sequence space to the extent that sequence numbers will not cycle through all 2**32 values before the segment data bound to those sequence numbers has been delivered and acknowledged by the receiver and all duplicate copies of the segments have "drained" from the internet. Without such an assumption, two distinct TCP segments could conceivably be assigned the same or overlapping sequence numbers, causing confusion at the receiver as to which data is new and which is old. Remember that each segment is bound to as many consecutive sequence numbers as there are octets of data and SYN or FIN flags in the segment.

Under normal conditions, TCPs keep track of the next sequence number to emit and the oldest awaiting acknowledgment so as to avoid mistakenly using a sequence number over before its first use has been acknowledged. This alone does not guarantee that old duplicate data is drained from the net, so the sequence space has been made very large to reduce the probability that a wandering duplicate will cause trouble upon arrival. At 2 megabits/sec. it takes 4.5 hours to use up 2**32 octets of sequence space. Since the maximum segment lifetime in the net is not likely to exceed a few tens of seconds, this is deemed ample protection for foreseeable nets, even if data rates escalate to 10's of megabits/sec. At 100 megabits/sec, the cycle time is 5.4 minutes which may be a little short, but still within reason.

The basic duplicate detection and sequencing algorithm in TCP can be defeated, however, if a source TCP does not have any memory of the sequence numbers it last used on a given connection. For example, if the TCP were to start all connections with sequence number 0, then upon crashing and restarting, a TCP might re-form an earlier connection (possibly after half-open connection resolution) and emit packets with sequence numbers identical to or overlapping with packets still in the network which were emitted on an earlier incarnation of the same connection. In the absence of knowledge about the sequence numbers used on a particular connection, the TCP specification recommends that the source delay for MSL seconds before emitting segments on the connection, to allow time for segments from the earlier connection incarnation to drain from the system.

Even hosts which can remember the time of day and used it to select initial sequence number values are not immune from this problem (i.e., even if time of day is used to select an initial sequence number for each new connection incarnation).

Suppose, for example, that a connection is opened starting with sequence number S. Suppose that this connection is not used much and that eventually the initial sequence number function (ISN(t)) takes on a value equal to the sequence number, say S1, of the last segment sent by this TCP on a particular connection. Now suppose, at this instant, the host crashes, recovers, and establishes a new incarnation of the connection. The initial sequence number chosen is S1 = ISN(t) -- last used sequence number on old incarnation of connection! If the recovery occurs quickly enough, any old duplicates in the net bearing sequence numbers in the neighborhood of S1 may arrive and be treated as new packets by the receiver of the new incarnation of the connection.

The problem is that the recovering host may not know for how long it crashed nor does it know whether there are still old duplicates in the system from earlier connection incarnations.

One way to deal with this problem is to deliberately delay emitting segments for one MSL after recovery from a crash- this is the "quiet time" specification. Hosts which prefer to avoid waiting are willing to risk possible confusion of old and new packets at a given destination may choose not to wait for the "quite time". Implementors may provide TCP users with the ability to select on a connection by connection basis whether to wait after a crash, or may informally implement the "quite time" for all connections. Obviously, even where a user selects to "wait," this is not necessary after the host has been "up" for at least MSL seconds.

To summarize: every segment emitted occupies one or more sequence numbers in the sequence space, the numbers occupied by a segment are "busy" or "in use" until MSL seconds have passed, upon crashing a block of space-time is occupied by the octets and SYN or FIN flags of the last emitted segment, if a new connection is started too soon and uses any of the sequence numbers in the space-time footprint of the last segment of the previous connection incarnation, there is a potential sequence number overlap area which could cause confusion at the receiver.

3.4. Establishing a connection

The "three-way handshake" is the procedure used to establish a connection. This procedure normally is initiated by one TCP and responded to by another TCP. The procedure also works if two TCP simultaneously initiate the procedure. When simultaneous attempt occurs, each TCP receives a "SYN" segment which carries no acknowledgment after it has sent a "SYN". Of course, the arrival of an old duplicate "SYN" segment can potentially make it appear, to the recipient, that a simultaneous connection initiation is in progress. Proper use of "reset" segments can disambiguate these cases.

Several examples of connection initiation follow. Although these examples do not show connection synchronization using data-carrying segments, this is perfectly legitimate, so long as the receiving TCP doesn't deliver the data to the user until it is clear the data is valid (i.e., the data must be buffered at the receiver until the connection reaches the ESTABLISHED state). The three-way handshake reduces the possibility of false connections. It is the implementation of a trade-off between memory and messages to provide information for this checking.

The simplest three-way handshake is shown in Figure 5 below. The figures should be interpreted in the following way. Each line is numbered for reference purposes. Right arrows (-->) indicate departure of a TCP segment from TCP A to TCP B, or arrival of a segment at B from A. Left arrows (<--), indicate the reverse. Ellipsis (...) indicates a segment which is still in the network (delayed). An "XXX" indicates a segment which is lost or rejected. Comments appear in parentheses. TCP states represent the state AFTER the departure or arrival of the segment (whose contents are shown in the center of each line). Segment contents are shown in abbreviated form, with sequence number, control flags, and ACK field. Other fields such as window, addresses, lengths, and text have been left out in the interest of clarity.

```
    TCP A                                                TCP B

1.  CLOSED                                               LISTEN

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

3.  ESTABLISHED <-- <SEQ=300><ACK=101><CTL=SYN,ACK>  <-- SYN-RECEIVED

4.  ESTABLISHED --> <SEQ=101><ACK=301><CTL=ACK>       --> ESTABLISHED

5.  ESTABLISHED --> <SEQ=101><ACK=301><CTL=ACK><DATA> --> ESTABLISHED
```

Figure 5: Basic 3-Way Handshake for Connection Synchronization

In line 2 of Figure 5, TCP A begins by sending a SYN segment indicating that it will use sequence numbers starting with sequence number 100. In line 3, TCP B sends a SYN and acknowledges the SYN it received from TCP A. Note that the acknowledgment field indicates TCP B is now expecting to hear sequence 101, acknowledging the SYN which occupied sequence 100.

At line 4, TCP A responds with an empty segment containing an ACK for TCP B's SYN; and in line 5, TCP A sends some data. Note that the sequence number of the segment in line 5 is the same as in line 4 because the ACK does not occupy sequence number space (if it did, we would wind up ACKing ACK's!).

Simultaneous initiation is only slightly more complex, as is shown in Figure 6. Each TCP cycles from CLOSED to SYN-SENT to SYN-RECEIVED to ESTABLISHED.

```
    TCP A                                                TCP B

1.  CLOSED                                               CLOSED

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

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

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

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

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

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

Figure 6: Simultaneous Connection Synchronization

A TCP MUST support simultaneous open attempts (MUST-10).

Note that a TCP implementation MUST keep track of whether a connection has reached SYN-RECEIVED state as the result of a passive OPEN or an active OPEN (MUST-11).

The principal reason for the three-way handshake is to prevent old duplicate connection initiations from causing confusion. To deal with this, a special control message, reset, has been devised. If the receiving TCP is in a non-synchronized state (i.e., SYN-SENT, SYN-RECEIVED), it returns to LISTEN on receiving an acceptable reset. If the TCP is in one of the synchronized states (ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT), it aborts the connection and informs its user. We discuss this latter case under "half-open" connections below.

```
       TCP A                                                TCP B

   1.  CLOSED                                               LISTEN

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

   3.  (duplicate) ... <SEQ=90><CTL=SYN>               --> SYN-RECEIVED

   4.  SYN-SENT    <-- <SEQ=300><ACK=91><CTL=SYN,ACK>  <-- SYN-RECEIVED

   5.  SYN-SENT    --> <SEQ=91><CTL=RST>               --> LISTEN

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

   7.  SYN-SENT    <-- <SEQ=400><ACK=101><CTL=SYN,ACK>  <-- SYN-RECEIVED

   8.  ESTABLISHED --> <SEQ=101><ACK=401><CTL=ACK>      --> ESTABLISHED
```

Figure 7: Recovery from Old Duplicate SYN

As a simple example of recovery from old duplicates, consider Figure 7. At line 3, an old duplicate SYN arrives at TCP B. TCP B cannot tell that this is an old duplicate, so it responds normally (line 4). TCP A detects that the ACK field is incorrect and returns a RST (reset) with its SEQ field selected to make the segment believable. TCP B, on receiving the RST, returns to the LISTEN state. When the original SYN (pun intended) finally arrives at line 6, the synchronization proceeds normally. If the SYN at line 6 had arrived before the RST, a more complex exchange might have occurred with RST's sent in both directions.

Half-Open Connections and Other Anomalies

An established connection is said to be "half-open" if one of the TCPs has closed or aborted the connection at its end without the knowledge of the other, or if the two ends of the connection have become desynchronized owing to a crash that resulted in loss of memory. Such connections will automatically become reset if an attempt is made to send data in either direction. However, half-open connections are expected to be unusual, and the recovery procedure is mildly involved.

If at site A the connection no longer exists, then an attempt by the user at site B to send any data on it will result in the site B TCP receiving a reset control message. Such a message indicates to the site B TCP that something is wrong, and it is expected to abort the connection.

Assume that two user processes A and B are communicating with one another when a crash occurs causing loss of memory to A's TCP. Depending on the operating system supporting A's TCP, it is likely that some error recovery mechanism exists. When the TCP is up again, A is likely to start again from the beginning or from a recovery point. As a result, A will probably try to OPEN the connection again or try to SEND on the connection it believes open. In the latter case, it receives the error message "connection not open" from the local (A's) TCP. In an attempt to establish the connection, A's TCP will send a segment containing SYN. This scenario leads to the example shown in Figure 8. After TCP A crashes, the user attempts to re-open the connection. TCP B, in the meantime, thinks the connection is open.

```
      TCP A                                           TCP B

  1.  (CRASH)                               (send 300,receive 100)

  2.  CLOSED                                           ESTABLISHED

  3.  SYN-SENT --> <SEQ=400><CTL=SYN>              --> (??)

  4.  (!!)     <-- <SEQ=300><ACK=100><CTL=ACK>     <-- ESTABLISHED

  5.  SYN-SENT --> <SEQ=100><CTL=RST>              --> (Abort!!)

  6.  SYN-SENT                                         CLOSED

  7.  SYN-SENT --> <SEQ=400><CTL=SYN>              -->
```

Figure 8: Half-Open Connection Discovery

When the SYN arrives at line 3, TCP B, being in a synchronized state, and the incoming segment outside the window, responds with an acknowledgment indicating what sequence it next expects to hear (ACK 100). TCP A sees that this segment does not acknowledge anything it sent and, being unsynchronized, sends a reset (RST) because it has detected a half-open connection. TCP B aborts at line 5. TCP A will continue to try to establish the connection; the problem is now reduced to the basic 3-way handshake of Figure 5.

An interesting alternative case occurs when TCP A crashes and TCP B tries to send data on what it thinks is a synchronized connection. This is illustrated in Figure 9. In this case, the data arriving at TCP A from TCP B (line 2) is unacceptable because no such connection exists, so TCP A sends a RST. The RST is acceptable so TCP B processes it and aborts the connection.

```
      TCP A                                                   TCP B

  1.  (CRASH)                                     (send 300,receive 100)

  2.  (??)    <-- <SEQ=300><ACK=100><DATA=10><CTL=ACK> <-- ESTABLISHED

  3.          --> <SEQ=100><CTL=RST>                   --> (ABORT!!)
```

Figure 9: Active Side Causes Half-Open Connection Discovery

In Figure 10, we find the two TCPs A and B with passive connections waiting for SYN. An old duplicate arriving at TCP B (line 2) stirs B into action. A SYN-ACK is returned (line 3) and causes TCP A to generate a RST (the ACK in line 3 is not acceptable). TCP B accepts the reset and returns to its passive LISTEN state.

```
    TCP A                                         TCP B

1.  LISTEN                                        LISTEN

2.       ... <SEQ=Z><CTL=SYN>                -->  SYN-RECEIVED

3.  (??) <-- <SEQ=X><ACK=Z+1><CTL=SYN,ACK>   <--  SYN-RECEIVED

4.       --> <SEQ=Z+1><CTL=RST>              -->  (return to LISTEN!)

5.  LISTEN                                        LISTEN
```

Figure 10: Old Duplicate SYN Initiates a Reset on two Passive Sockets

A variety of other cases are possible, all of which are accounted for by the following rules for RST generation and processing.

Reset Generation

As a general rule, reset (RST) must be sent whenever a segment arrives which apparently is not intended for the current connection. A reset must not be sent if it is not clear that this is the case.

There are three groups of states:

1. If the connection does not exist (CLOSED) then a reset is sent in response to any incoming segment except another reset. In particular, SYNs addressed to a non-existent connection are rejected by this means.

If the incoming segment has the ACK bit set, the reset takes its sequence number from the ACK field of the segment, otherwise the reset has sequence number zero and the ACK field is set to the sum of the sequence number and segment length of the incoming segment. The connection remains in the CLOSED state.

2. If the connection is in any non-synchronized state (LISTEN, SYN-SENT, SYN-RECEIVED), and the incoming segment acknowledges something not yet sent (the segment carries an unacceptable ACK), or if an incoming segment has a security level or compartment which does not exactly match the level and compartment requested for the connection, a reset is sent.

If the incoming segment has an ACK field, the reset takes its sequence number from the ACK field of the segment, otherwise the reset has sequence number zero and the ACK field is set to the sum of the sequence number and segment length of the incoming segment. The connection remains in the same state.

3. If the connection is in a synchronized state (ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT), any unacceptable segment (out of window sequence number or unacceptable acknowledgment number) must elicit only an empty acknowledgment segment containing the current send-sequence number and an acknowledgment indicating the next sequence number expected to be received, and the connection remains in the same state.

If an incoming segment has a security level, or compartment which does not exactly match the level and compartment requested for the connection, a reset is sent and the connection goes to the CLOSED state. The reset takes its sequence number from the ACK field of the incoming segment.

Reset Processing

In all states except SYN-SENT, all reset (RST) segments are validated by checking their SEQ-fields. A reset is valid if its sequence number is in the window. In the SYN-SENT state (a RST received in response to an initial SYN), the RST is acceptable if the ACK field acknowledges the SYN.

The receiver of a RST first validates it, then changes state. If the receiver was in the LISTEN state, it ignores it. If the receiver was in SYN-RECEIVED state and had previously been in the LISTEN state, then the receiver returns to the LISTEN state, otherwise the receiver aborts the connection and goes to the CLOSED state. If the receiver was in any other state, it aborts the connection and advises the user and goes to the CLOSED state.

TCP SHOULD allow a received RST segment to include data (SHLD-2).

3.5. Closing a Connection

CLOSE is an operation meaning "I have no more data to send." The notion of closing a full-duplex connection is subject to ambiguous interpretation, of course, since it may not be obvious how to treat the receiving side of the connection. We have chosen to treat CLOSE in a simplex fashion. The user who CLOSEs may continue to RECEIVE until he is told that the other side has CLOSED also. Thus, a program could initiate several SENDs followed by a CLOSE, and then continue to RECEIVE until signaled that a RECEIVE failed because the other side has CLOSED. We assume that the TCP will signal a user, even if no RECEIVEs are outstanding, that the other side has closed, so the user can terminate his side gracefully. A TCP will reliably deliver all buffers SENT before the connection was CLOSED so a user who expects no data in return need only wait to hear the connection was CLOSED successfully to know that all his data was received at the destination TCP. Users must keep reading connections they close for sending until the TCP says no more data.

There are essentially three cases:

1) The user initiates by telling the TCP to CLOSE the connection

2) The remote TCP initiates by sending a FIN control signal

3) Both users CLOSE simultaneously

Case 1: Local user initiates the close

In this case, a FIN segment can be constructed and placed on the outgoing segment queue. No further SENDs from the user will be accepted by the TCP, and it enters the FIN-WAIT-1 state. RECEIVEs are allowed in this state. All segments preceding and including FIN will be retransmitted until acknowledged. When the other TCP has both acknowledged the FIN and sent a FIN of its own, the first TCP can ACK this FIN. Note that a TCP receiving a FIN will ACK but not send its own FIN until its user has CLOSED the connection also.

Case 2: TCP receives a FIN from the network

If an unsolicited FIN arrives from the network, the receiving TCP can ACK it and tell the user that the connection is closing. The user will respond with a CLOSE, upon which the TCP can send a FIN to the other TCP after sending any remaining data. The TCP then waits until its own FIN is acknowledged whereupon it deletes the connection. If an ACK is not forthcoming, after the user timeout the connection is aborted and the user is told.

Case 3: both users close simultaneously

A simultaneous CLOSE by users at both ends of a connection causes FIN segments to be exchanged. When all segments preceding the FINs have been processed and acknowledged, each TCP can ACK the FIN it has received. Both will, upon receiving these ACKs, delete the connection.

```
    TCP A                                                    TCP B

1.  ESTABLISHED                                              ESTABLISHED

2.  (Close)
    FIN-WAIT-1  --> <SEQ=100><ACK=300><CTL=FIN,ACK>  --> CLOSE-WAIT

3.  FIN-WAIT-2  <-- <SEQ=300><ACK=101><CTL=ACK>      <-- CLOSE-WAIT

4.                                                       (Close)
    TIME-WAIT   <-- <SEQ=300><ACK=101><CTL=FIN,ACK>  <-- LAST-ACK

5.  TIME-WAIT   --> <SEQ=101><ACK=301><CTL=ACK>      --> CLOSED

6.  (2 MSL)
    CLOSED
```

Figure 11: Normal Close Sequence

```
    TCP A                                                TCP B

1.  ESTABLISHED                                          ESTABLISHED

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

3.  CLOSING     --> <SEQ=101><ACK=301><CTL=ACK>      ... CLOSING
                <-- <SEQ=301><ACK=101><CTL=ACK>      <--
                ... <SEQ=101><ACK=301><CTL=ACK>      -->

4.  TIME-WAIT                                            TIME-WAIT
    (2 MSL)                                              (2 MSL)
    CLOSED                                               CLOSED
```

Figure 12: Simultaneous Close Sequence

A TCP connection may terminate in two ways: (1) the normal TCP close sequence using a FIN handshake, and (2) an "abort" in which one or more RST segments are sent and the connection state is immediately discarded. If the local TCP connection is closed by the remote side due to a FIN or RST received from the remote side, then the local application MUST be informed whether it closed normally or was aborted (MUST-12).

3.5.1. Half-Closed Connections

The normal TCP close sequence delivers buffered data reliably in both directions. Since the two directions of a TCP connection are closed independently, it is possible for a connection to be "half closed," i.e., closed in only one direction, and a host is permitted to continue sending data in the open direction on a half-closed connection.

A host MAY implement a "half-duplex" TCP close sequence, so that an application that has called CLOSE cannot continue to read data from the connection (MAY-1). If such a host issues a CLOSE call while received data is still pending in TCP, or if new data is received after CLOSE is called, its TCP SHOULD send a RST to show that data was lost (SHLD-3). See [17] section 2.17 for discussion.

When a connection is closed actively, it MUST linger in TIME-WAIT state for a time 2xMSL (Maximum Segment Lifetime) (MUST-13). However, it MAY accept a new SYN from the remote TCP to reopen the connection directly from TIME-WAIT state (MAY-2), if it:

(1) assigns its initial sequence number for the new connection to be larger than the largest sequence number it used on the previous connection incarnation, and

(2) returns to TIME-WAIT state if the SYN turns out to be an old duplicate.

When the TCP Timestamp options are available, an improved algorithm is described in [30] in order to support higher connection establishment rates. This algorithm for reducing TIME-WAIT is a Best Current Practice that SHOULD be implemented, since timestamp options are commonly used, and using them to reduce TIME-WAIT provides benefits for busy Internet servers (SHLD-4).

3.6. Precedence and Security

The IPv4 specification [1] includes a precedence value in the (now obsoleted) Type of Service field (TOS) field. It was modified in [15], and then obsoleted by the definition of Differentiated Services (DiffServ) [5]. Setting and conveying TOS between the network layer, TCP, and applications is obsolete, and replaced by DiffServ in the current TCP specification.

In DiffServ the former precedence values are treated as Class Selector codepoints, and methods for compatible treatment are described in the DiffServ architecture. The RFC 793/1122 TCP specification includes logic intending to have connections use the highest precedence requested by either endpoint application, and to keep the precedence consistent throughout a connection. This logic from the obsolete TOS is not applicable for DiffServ, and should not be included in TCP implementations, though changes to DiffServ values within a connection are discouraged. For discussion of this, see RFC 7657 (sec 5.1, 5.3, and 6) [38].

The obsoleted TOS processing rules in TCP assumed bidirectional (or symmetric) precedence values used on a connection, but the DiffServ architecture is asymmetric. Problems with the old TCP logic in this regard were described in [18] and the solution described is to ignore IP precedence in TCP. Since RFC 2873 is a Standards Track document (although not marked as updating RFC 793), current implementations are expected to be robust to these conditions. Note that the DiffServ field value used in each direction is a part of the interface between TCP and the network layer, and values in use can be indicated both ways between TCP and the application.

The IP security option (IPSO) and compartment defined in [1] was refined in RFC 1038 that was later obsoleted by RFC 1108. The Commercial IP Security Option (CIPSO) is defined in FIPS-188, and is supported by some vendors and operating systems. RFC 1108 is now Historic, though RFC 791 itself has not been updated to remove the IP security option. For IPv6, a similar option (CALIPSO) has been defined [24]. RFC 793 includes logic that includes the IP security/compartment information in treatment of TCP segments. References to the IP "security/compartment" in this document may be relevant for Multi-Level Secure (MLS) system implementers, but can be ignored for non-MLS implementations, consistent with running code on the Internet. See Appendix A.1 for further discussion. Note that RFC 5570 describes some MLS networking scenarios where IPSO, CIPSO, or CALIPSO may be used. In these special cases, TCP implementers should see section 7.3.1 of RFC 5570, and follow the guidance in that document on the relation between IP security.

3.7. Segmentation

The term "segmentation" refers to the activity TCP performs when ingesting a stream of bytes from a sending application and packetizing that stream of bytes into TCP segments. Individual TCP segments often do not correspond one-for-one to individual send (or socket write) calls from the application. Applications may perform writes at the granularity of messages in the upper layer protocol, but TCP guarantees no boundary coherence between the TCP segments sent and received versus user application data read or write buffer boundaries. In some specific protocols, such as RDMA using DDP and MPA [22], there are performance optimizations possible when the relation between TCP segments and application data units can be controlled, and MPA includes a specific mechanism for detecting and verifying this relationship between TCP segments and application message data strcutures, but this is specific to applications like RDMA. In general, multiple goals influence the sizing of TCP segments created by a TCP implementation.

Goals driving the sending of larger segments include:

- o Reducing the number of packets in flight within the network.
- o Increasing processing efficiency and potential performance by enabling a smaller number of interrupts and inter-layer interactions.
- o Limiting the overhead of TCP headers.

Note that the performance benefits of sending larger segments may decrease as the size increases, and there may be boundaries where advantages are reversed. For instance, on some machines 1025 bytes within a segment could lead to worse performance than 1024 bytes, due purely to data alignment on copy operations.

Goals driving the sending of smaller segments include:

- o Avoiding sending segments larger than the smallest MTU within an IP network path, because this results in either packet loss or fragmentation. Making matters worse, some firewalls or middleboxes may drop fragmented packets or ICMP messages related related to fragmentation.

- o Preventing delays to the application data stream, especially when TCP is waiting on the application to generate more data, or when the application is waiting on an event or input from its peer in order to generate more data.

- o Enabling "fate sharing" between TCP segments and lower-layer data units (e.g. below IP, for links with cell or frame sizes smaller than the IP MTU).

Towards meeting these competing sets of goals, TCP includes several mechanisms, including the Maximum Segment Size option, Path MTU Discovery, the Nagle algorithm, and support for IPv6 Jumbograms, as discussed in the following subsections.

3.7.1. Maximum Segment Size Option

TCP MUST implement both sending and receiving the MSS option (MUST-14).

TCP SHOULD send an MSS option in every SYN segment when its receive MSS differs from the default 536 for IPv4 or 1220 for IPv6 (SHLD-5), and MAY send it always (MAY-3).

If an MSS option is not received at connection setup, TCP MUST assume a default send MSS of 536 (576-40) for IPv4 or 1220 (1280 - 60) for IPv6 (MUST-15).

The maximum size of a segment that TCP really sends, the "effective send MSS," MUST be the smaller (MUST-16) of the send MSS (which reflects the available reassembly buffer size at the remote host, the EMTU_R [14]) and the largest transmission size permitted by the IP layer (EMTU_S [14]):

```
Eff.snd.MSS =

    min(SendMSS+20, MMS_S) - TCPhdrsize - IPoptionsize
```

where:

- o SendMSS is the MSS value received from the remote host, or the default 536 for IPv4 or 1220 for IPv6, if no MSS option is received.

- o MMS_S is the maximum size for a transport-layer message that TCP may send.

- o TCPhdrsize is the size of the fixed TCP header and any options. This is 20 in the (rare) case that no options are present, but may be larger if TCP options are to be sent. Note that some options may not be included on all segments, but that for each segment sent, the sender should adjust the data length accordingly, within the Eff.snd.MSS.

- o IPoptionsize is the size of any IP options associated with a TCP connection. Note that some options may not be included on all packets, but that for each segment sent, the sender should adjust the data length accordingly, within the Eff.snd.MSS.

The MSS value to be sent in an MSS option should be equal to the effective MTU minus the fixed IP and TCP headers. By ignoring both IP and TCP options when calculating the value for the MSS option, if there are any IP or TCP options to be sent in a packet, then the sender must decrease the size of the TCP data accordingly. RFC 6691 [33] discusses this in greater detail.

The MSS value to be sent in an MSS option must be less than or equal to:

MMS_R - 20

where MMS_R is the maximum size for a transport-layer message that can be received (and reassembled at the IP layer). TCP obtains MMS_R and MMS_S from the IP layer; see the generic call GET_MAXSIZES in Section 3.4 of RFC 1122. These are defined in terms of their IP MTU equivalents, EMTU_R and EMTU_S [14].

When TCP is used in a situation where either the IP or TCP headers are not fixed, the sender must reduce the amount of TCP data in any given packet by the number of octets used by the IP and TCP options. This has been a point of confusion historically, as explained in RFC 6691, Section 3.1.

3.7.2. Path MTU Discovery

A TCP implementation may be aware of the MTU on directly connected links, but will rarely have insight about MTUs across an entire network path. For IPv4, RFC 1122 provides an IP-layer recommendation on the default effective MTU for sending to be less than or equal to 576 for destinations not directly connected. For IPv6, this would be 1280. In all cases, however, implementation of Path MTU Discovery (PMTUD) and Packetization Layer Path MTU Discovery (PLPMTUD) is strongly recommended in order for TCP to improve segmentation decisions. Both PMTUD and PLPMTUD help TCP choose segment sizes that avoid both on-path (for IPv4) and source fragmentation (IPv4 and IPv6).

PMTUD for IPv4 [2] or IPv6 [3] is implemented in conjunction between TCP, IP, and ICMP protocols. It relies both on avoiding source fragmentation and setting the IPv4 DF (don't fragment) flag, the latter to inhibit on-path fragmentation. It relies on ICMP errors from routers along the path, whenever a segment is too large to traverse a link. Several adjustments to a TCP implementation with PMTUD are described in RFC 2923 in order to deal with problems experienced in practice [7]. PLPMTUD [19] is a Standards Track improvement to PMTUD that relaxes the requirement for ICMP support across a path, and improves performance in cases where ICMP is not consistently conveyed, but still tries to avoid source fragmentation. The mechanisms in all four of these RFCs are recommended to be included in TCP implementations.

The TCP MSS option specifies an upper bound for the size of packets that can be received. Hence, setting the value in the MSS option too small can impact the ability for PMTUD or PLPMTUD to find a larger path MTU. RFC 1191 discusses this implication of many older TCP implementations setting MSS to 536 for non-local destinations, rather than deriving it from the MTUs of connected interfaces as recommended.

3.7.3. Interfaces with Variable MTU Values

The effective MTU can sometimes vary, as when used with variable compression, e.g., RObust Header Compression (ROHC) [26]. It is tempting for TCP to want to advertise the largest possible MSS, to support the most efficient use of compressed payloads. Unfortunately, some compression schemes occasionally need to transmit full headers (and thus smaller payloads) to resynchronize state at their endpoint compressors/decompressors. If the largest MTU is used to calculate the value to advertise in the MSS option, TCP retransmission may interfere with compressor resynchronization.

As a result, when the effective MTU of an interface varies, TCP SHOULD use the smallest effective MTU of the interface to calculate the value to advertise in the MSS option (SHLD-6).

3.7.4. Nagle Algorithm

The "Nagle algorithm" was described in RFC 896 [13] and was recommended in RFC 1122 [14] for mitigation of an early problem of too many small packets being generated. It has been implemented in most current TCP code bases, sometimes with minor variations (see Appendix A.3).

If there is unacknowledged data (i.e., SND.NXT > SND.UNA), then the sending TCP buffers all user data (regardless of the PSH bit), until the outstanding data has been acknowledged or until the TCP can send a full-sized segment (Eff.snd.MSS bytes).

A TCP SHOULD implement the Nagle Algorithm to coalesce short segments (SHLD-7). However, there MUST be a way for an application to disable the Nagle algorithm on an individual connection (MUST-17). In all cases, sending data is also subject to the limitation imposed by the Slow Start algorithm [25].

3.7.5. IPv6 Jumbograms

In order to support TCP over IPv6 jumbograms, implementations need to be able to send TCP segments larger than the 64KB limit that the MSS option can convey. RFC 2675 [6] defines that an MSS value of 65,535 bytes is to be treated as infinity, and Path MTU Discovery [3] is used to determine the actual MSS.

3.8. Data Communication

Once the connection is established data is communicated by the exchange of segments. Because segments may be lost due to errors (checksum test failure), or network congestion, TCP uses retransmission (after a timeout) to ensure delivery of every segment. Duplicate segments may arrive due to network or TCP retransmission. As discussed in the section on sequence numbers the TCP performs certain tests on the sequence and acknowledgment numbers in the segments to verify their acceptability.

The sender of data keeps track of the next sequence number to use in the variable SND.NXT. The receiver of data keeps track of the next sequence number to expect in the variable RCV.NXT. The sender of data keeps track of the oldest unacknowledged sequence number in the variable SND.UNA. If the data flow is momentarily idle and all data sent has been acknowledged then the three variables will be equal.

When the sender creates a segment and transmits it the sender advances SND.NXT. When the receiver accepts a segment it advances RCV.NXT and sends an acknowledgment. When the data sender receives an acknowledgment it advances SND.UNA. The extent to which the values of these variables differ is a measure of the delay in the communication. The amount by which the variables are advanced is the length of the data and SYN or FIN flags in the segment. Note that once in the ESTABLISHED state all segments must carry current acknowledgment information.

The CLOSE user call implies a push function, as does the FIN control flag in an incoming segment.

3.8.1. Retransmission Timeout

Because of the variability of the networks that compose an internetwork system and the wide range of uses of TCP connections the retransmission timeout (RTO) must be dynamically determined.

The RTO MUST be computed according to the algorithm in [9], including Karn's algorithm for taking RTT samples (MUST-18).

RFC 793 contains an early example procedure for computing the RTO. This was then replaced by the algorithm described in RFC 1122, and subsequently updated in RFC 2988, and then again in RFC 6298.

If a retransmitted packet is identical to the original packet (which implies not only that the data boundaries have not changed, but also that the window and acknowledgment fields of the header have not changed), then the same IP Identification field MAY be used (see Section 3.2.1.5 of RFC 1122) (MAY-4).

3.8.2. TCP Congestion Control

RFC 1122 required implementation of Van Jacobson's congestion control algorithm combining slow start with congestion avoidance. RFC 2581 provided IETF Standards Track description of this, along with fast retransmit and fast recovery. RFC 5681 is the current description of these algorithms and is the current standard for TCP congestion control.

A TCP MUST implement RFC 5681 (MUST-19).

Explicit Congestion Notification (ECN) was defined in RFC 3168 and is an IETF Standards Track enhancement that has many benefits [39].

A TCP SHOULD implement ECN as described in RFC 3168 (SHLD-8).

3.8.3. TCP Connection Failures

Excessive retransmission of the same segment by TCP indicates some failure of the remote host or the Internet path. This failure may be of short or long duration. The following procedure MUST be used to handle excessive retransmissions of data segments (MUST-20):

(a) There are two thresholds R1 and R2 measuring the amount of retransmission that has occurred for the same segment. R1 and R2 might be measured in time units or as a count of retransmissions.

(b) When the number of transmissions of the same segment reaches or exceeds threshold R1, pass negative advice (see [14] Section 3.3.1.4) to the IP layer, to trigger dead-gateway diagnosis.

(c) When the number of transmissions of the same segment reaches a threshold R2 greater than R1, close the connection.

(d) An application MUST (MUST-21) be able to set the value for R2 for a particular connection. For example, an interactive application might set R2 to "infinity," giving the user control over when to disconnect.

(e) TCP SHOULD inform the application of the delivery problem (unless such information has been disabled by the application; see Asynchronous Reports section), when R1 is reached and before R2 (SHLD-9). This will allow a remote login (User Telnet) application program to inform the user, for example.

The value of R1 SHOULD correspond to at least 3 retransmissions, at the current RTO (SHLD-10). The value of R2 SHOULD correspond to at least 100 seconds (SHLD-11).

An attempt to open a TCP connection could fail with excessive retransmissions of the SYN segment or by receipt of a RST segment or an ICMP Port Unreachable. SYN retransmissions MUST be handled in the general way just described for data retransmissions, including notification of the application layer.

However, the values of R1 and R2 may be different for SYN and data segments. In particular, R2 for a SYN segment MUST be set large enough to provide retransmission of the segment for at least 3 minutes. The application can close the connection (i.e., give up on the open attempt) sooner, of course.

3.8.4. TCP Keep-Alives

Implementors MAY include "keep-alives" in their TCP implementations (MAY-5), although this practice is not universally accepted. If keep-alives are included, the application MUST be able to turn them on or off for each TCP connection (MUST-24), and they MUST default to off (MUST-25).

Keep-alive packets MUST only be sent when no data or acknowledgement packets have been received for the connection within an interval (MUST-26). This interval MUST be configurable (MUST-27) and MUST default to no less than two hours (MUST-28).

It is extremely important to remember that ACK segments that contain no data are not reliably transmitted by TCP. Consequently, if a keep-alive mechanism is implemented it MUST NOT interpret failure to respond to any specific probe as a dead connection (MUST-29).

An implementation SHOULD send a keep-alive segment with no data (SHLD-12); however, it MAY be configurable to send a keep-alive segment containing one garbage octet (MAY-6), for compatibility with erroneous TCP implementations.

3.8.5. The Communication of Urgent Information

As a result of implementation differences and middlebox interactions, new applications SHOULD NOT employ the TCP urgent mechanism (SHLD-13). However, TCP implementations MUST still include support for the urgent mechanism (MUST-30). Details can be found in RFC 6093 [29].

The objective of the TCP urgent mechanism is to allow the sending user to stimulate the receiving user to accept some urgent data and to permit the receiving TCP to indicate to the receiving user when all the currently known urgent data has been received by the user.

This mechanism permits a point in the data stream to be designated as the end of urgent information. Whenever this point is in advance of the receive sequence number (RCV.NXT) at the receiving TCP, that TCP must tell the user to go into "urgent mode"; when the receive sequence number catches up to the urgent pointer, the TCP must tell user to go into "normal mode". If the urgent pointer is updated while the user is in "urgent mode", the update will be invisible to the user.

The method employs a urgent field which is carried in all segments transmitted. The URG control flag indicates that the urgent field is meaningful and must be added to the segment sequence number to yield the urgent pointer. The absence of this flag indicates that there is no urgent data outstanding.

To send an urgent indication the user must also send at least one data octet. If the sending user also indicates a push, timely delivery of the urgent information to the destination process is enhanced.

A TCP MUST support a sequence of urgent data of any length (MUST-31). [14]

The urgent pointer MUST point to the sequence number of the octet following the urgent data (MUST-62).

A TCP MUST (MUST-32) inform the application layer asynchronously whenever it receives an Urgent pointer and there was previously no pending urgent data, or whenvever the Urgent pointer advances in the data stream. There MUST (MUST-33) be a way for the application to learn how much urgent data remains to be read from the connection, or at least to determine whether or not more urgent data remains to be read. [14]

3.8.6. Managing the Window

The window sent in each segment indicates the range of sequence numbers the sender of the window (the data receiver) is currently prepared to accept. There is an assumption that this is related to the currently available data buffer space available for this connection.

The sending TCP packages the data to be transmitted into segments which fit the current window, and may repackage segments on the retransmission queue. Such repackaging is not required, but may be helpful.

In a connection with a one-way data flow, the window information will be carried in acknowledgment segments that all have the same sequence number so there will be no way to reorder them if they arrive out of order. This is not a serious problem, but it will allow the window information to be on occasion temporarily based on old reports from the data receiver. A refinement to avoid this problem is to act on the window information from segments that carry the highest acknowledgment number (that is segments with acknowledgment number equal or greater than the highest previously received).

Indicating a large window encourages transmissions. If more data arrives than can be accepted, it will be discarded. This will result in excessive retransmissions, adding unnecessarily to the load on the network and the TCPs. Indicating a small window may restrict the transmission of data to the point of introducing a round trip delay between each new segment transmitted.

The mechanisms provided allow a TCP to advertise a large window and to subsequently advertise a much smaller window without having accepted that much data. This, so called "shrinking the window," is strongly discouraged. The robustness principle [14] dictates that TCPs will not shrink the window themselves, but will be prepared for such behavior on the part of other TCPs.

A TCP receiver SHOULD NOT shrink the window, i.e., move the right window edge to the left (SHLD-14). However, a sending TCP MUST be robust against window shrinking, which may cause the "useable window" (see Section 3.8.6.2.1) to become negative (MUST-34).

If this happens, the sender SHOULD NOT send new data (SHLD-15), but SHOULD retransmit normally the old unacknowledged data between SND.UNA and SND.UNA+SND.WND (SHLD-16). The sender MAY also retransmit old data beyond SND.UNA+SND.WND (MAY-7), but SHOULD NOT time out the connection if data beyond the right window edge is not acknowledged (SHLD-17). If the window shrinks to zero, the TCP MUST probe it in the standard way (described below) (MUST-35).

3.8.6.1. Zero Window Probing

The sending TCP must be prepared to accept from the user and send at least one octet of new data even if the send window is zero. The sending TCP must regularly retransmit to the receiving TCP even when the window is zero, in order to "probe" the window. Two minutes is recommended for the retransmission interval when the window is zero. This retransmission is essential to guarantee that when either TCP has a zero window the re-opening of the window will be reliably reported to the other. This is referred to as Zero-Window Probing (ZWP) in other documents.

Probing of zero (offered) windows MUST be supported (MUST-36).

A TCP MAY keep its offered receive window closed indefinitely (MAY-8). As long as the receiving TCP continues to send acknowledgments in response to the probe segments, the sending TCP MUST allow the connection to stay open (MUST-37). This enables TCP to function in scenarios such as the "printer ran out of paper" situation described in Section 4.2.2.17 of RFC1122. The behavior is subject to the implementation's resource management concerns, as noted in [31].

When the receiving TCP has a zero window and a segment arrives it must still send an acknowledgment showing its next expected sequence number and current window (zero).

The transmitting host SHOULD send the first zero-window probe when a zero window has existed for the retransmission timeout period (SHLD-29) (see Section 3.8.1), and SHOULD increase exponentially the interval between successive probes (SHLD-30).

3.8.6.2. Silly Window Syndrome Avoidance

The "Silly Window Syndrome" (SWS) is a stable pattern of small incremental window movements resulting in extremely poor TCP performance. Algorithms to avoid SWS are described below for both the sending side and the receiving side. RFC 1122 contains more detailed discussion of the SWS problem. Note that the Nagle algorithm and the sender SWS avoidance algorithm play complementary roles in improving performance. The Nagle algorithm discourages sending tiny segments when the data to be sent increases in small increments, while the SWS avoidance algorithm discourages small segments resulting from the right window edge advancing in small increments.

3.8.6.2.1. Sender's Algorithm - When to Send Data

A TCP MUST include a SWS avoidance algorithm in the sender (MUST-38).

The Nagle algorithm from Section 3.7.4 additionally describes how to coalesce short segments.

The sender's SWS avoidance algorithm is more difficult than the receivers's, because the sender does not know (directly) the receiver's total buffer space RCV.BUFF. An approach which has been found to work well is for the sender to calculate Max(SND.WND), the maximum send window it has seen so far on the connection, and to use this value as an estimate of RCV.BUFF. Unfortunately, this can only be an estimate; the receiver may at any time reduce the size of RCV.BUFF. To avoid a resulting deadlock, it is necessary to have a timeout to force transmission of data, overriding the SWS avoidance algorithm. In practice, this timeout should seldom occur.

The "useable window" is:

    U = SND.UNA + SND.WND - SND.NXT

i.e., the offered window less the amount of data sent but not acknowledged. If D is the amount of data queued in the sending TCP but not yet sent, then the following set of rules is recommended.

Send data:

(1) if a maximum-sized segment can be sent, i.e, if:

min(D,U) >= Eff.snd.MSS;

(2) or if the data is pushed and all queued data can be sent now, i.e., if:

[SND.NXT = SND.UNA and] PUSHED and D <= U

(the bracketed condition is imposed by the Nagle algorithm);

(3) or if at least a fraction Fs of the maximum window can be sent, i.e., if:

[SND.NXT = SND.UNA and]

min(D.U) >= Fs * Max(SND.WND);

(4) or if data is PUSHed and the override timeout occurs.

Here Fs is a fraction whose recommended value is 1/2. The override timeout should be in the range 0.1 - 1.0 seconds. It may be convenient to combine this timer with the timer used to probe zero windows (Section Section 3.8.6.1).

3.8.6.2.2. Receiver's Algorithm - When to Send a Window Update

A TCP MUST include a SWS avoidance algorithm in the receiver (MUST-39).

The receiver's SWS avoidance algorithm determines when the right window edge may be advanced; this is customarily known as "updating the window". This algorithm combines with the delayed ACK algorithm (see Section 3.8.6.3) to determine when an ACK segment containing the current window will really be sent to the receiver.

The solution to receiver SWS is to avoid advancing the right window edge RCV.NXT+RCV.WND in small increments, even if data is received from the network in small segments.

Suppose the total receive buffer space is RCV.BUFF. At any given moment, RCV.USER octets of this total may be tied up with data that has been received and acknowledged but which the user process has not yet consumed. When the connection is quiescent, RCV.WND = RCV.BUFF and RCV.USER = 0.

Keeping the right window edge fixed as data arrives and is acknowledged requires that the receiver offer less than its full buffer space, i.e., the receiver must specify a RCV.WND that keeps RCV.NXT+RCV.WND constant as RCV.NXT increases. Thus, the total buffer space RCV.BUFF is generally divided into three parts:

```
                |<------- RCV.BUFF ---------------->|
                     1             2            3
            ----|---------|------------------|------|----
                       RCV.NXT               ^
                                          (Fixed)

            1 - RCV.USER =  data received but not yet consumed;
            2 - RCV.WND =   space advertised to sender;
            3 - Reduction = space available but not yet
                            advertised.
```

The suggested SWS avoidance algorithm for the receiver is to keep RCV.NXT+RCV.WND fixed until the reduction satisfies:

```
             RCV.BUFF - RCV.USER - RCV.WND  >=

                    min( Fr * RCV.BUFF, Eff.snd.MSS )
```

where Fr is a fraction whose recommended value is 1/2, and Eff.snd.MSS is the effective send MSS for the connection (see Section 3.7.1). When the inequality is satisfied, RCV.WND is set to RCV.BUFF-RCV.USER.

Note that the general effect of this algorithm is to advance RCV.WND in increments of Eff.snd.MSS (for realistic receive buffers: Eff.snd.MSS < RCV.BUFF/2). Note also that the receiver must use its own Eff.snd.MSS, assuming it is the same as the sender’s.

3.8.6.3. Delayed Acknowledgements - When to Send an ACK Segment

A host that is receiving a stream of TCP data segments can increase efficiency in both the Internet and the hosts by sending fewer than one ACK (acknowledgment) segment per data segment received; this is known as a "delayed ACK".

A TCP SHOULD implement a delayed ACK (SHLD-18), but an ACK should not be excessively delayed; in particular, the delay MUST be less than 0.5 seconds (MUST-40), and in a stream of full-sized segments there SHOULD be an ACK for at least every second segment (SHLD-19).

Excessive delays on ACK's can disturb the round-trip timing and packet "clocking" algorithms.

3.9. Interfaces

There are of course two interfaces of concern: the user/TCP interface and the TCP/lower-level interface. We have a fairly elaborate model of the user/TCP interface, but the interface to the lower level protocol module is left unspecified here, since it will be specified in detail by the specification of the lower level protocol. For the case that the lower level is IP we note some of the parameter values that TCPs might use.

3.9.1. User/TCP Interface

The following functional description of user commands to the TCP is, at best, fictional, since every operating system will have different facilities. Consequently, we must warn readers that different TCP implementations may have different user interfaces. However, all TCPs must provide a certain minimum set of services to guarantee that all TCP implementations can support the same protocol hierarchy. This section specifies the functional interfaces required of all TCP implementations.

TCP User Commands

The following sections functionally characterize a USER/TCP interface. The notation used is similar to most procedure or function calls in high level languages, but this usage is not meant to rule out trap type service calls.

The user commands described below specify the basic functions the TCP must perform to support interprocess communication. Individual implementations must define their own exact format, and may provide combinations or subsets of the basic functions in single calls. In particular, some implementations may wish to automatically OPEN a connection on the first SEND or RECEIVE issued by the user for a given connection.

In providing interprocess communication facilities, the TCP must not only accept commands, but must also return information to the processes it serves. The latter consists of:

(a) general information about a connection (e.g., interrupts, remote close, binding of unspecified foreign socket).

(b) replies to specific user commands indicating success or various types of failure.

Open

Format: OPEN (local port, foreign socket, active/passive [, timeout] [, DiffServ field] [, security/compartment] [local IP address,] [, options]) -> local connection name

We assume that the local TCP is aware of the identity of the processes it serves and will check the authority of the process to use the connection specified. Depending upon the implementation of the TCP, the local network and TCP identifiers for the source address will either be supplied by the TCP or the lower level protocol (e.g., IP). These considerations are the result of concern about security, to the extent that no TCP be able to masquerade as another one, and so on. Similarly, no process can masquerade as another without the collusion of the TCP.

If the active/passive flag is set to passive, then this is a call to LISTEN for an incoming connection. A passive open may have either a fully specified foreign socket to wait for a particular connection or an unspecified foreign socket to wait for any call. A fully specified passive call can be made active by the subsequent execution of a SEND.

A transmission control block (TCB) is created and partially filled in with data from the OPEN command parameters.

Every passive OPEN call either creates a new connection record in LISTEN state, or it returns an error; it MUST NOT affect any previously created connection record (MUST-41).

A TCP that supports multiple concurrent users MUST provide an OPEN call that will functionally allow an application to LISTEN on a port while a connection block with the same local port is in SYN-SENT or SYN-RECEIVED state (MUST-42).

On an active OPEN command, the TCP will begin the procedure to synchronize (i.e., establish) the connection at once.

The timeout, if present, permits the caller to set up a timeout for all data submitted to TCP. If data is not successfully delivered to the destination within the timeout period, the TCP will abort the connection. The present global default is five minutes.

The TCP or some component of the operating system will verify the users authority to open a connection with the specified DiffServ field value or security/compartment. The absence of a

DiffServ field value or security/compartment specification in the OPEN call indicates the default values must be used.

TCP will accept incoming requests as matching only if the security/compartment information is exactly the same as that requested in the OPEN call.

The DiffServ field value indicated by the user only impacts outgoing packets, may be altered en route through the network, and has no direct bearing or relation to received packets.

A local connection name will be returned to the user by the TCP. The local connection name can then be used as a short hand term for the connection defined by the <local socket, foreign socket> pair.

The optional "local IP address" parameter MUST be supported to allow the specification of the local IP address (MUST-43). This enables applications that need to select the local IP address used when multihoming is present.

A passive OPEN call with a specified "local IP address" parameter will await an incoming connection request to that address. If the parameter is unspecified, a passive OPEN will await an incoming connection request to any local IP address, and then bind the local IP address of the connection to the particular address that is used.

For an active OPEN call, a specified "local IP address" parameter will be used for opening the connection. If the parameter is unspecified, the host will choose an appropriate local IP address (see RFC 1122 section 3.3.4.2).

If an application on a multihomed host does not specify the local IP address when actively opening a TCP connection, then the TCP MUST ask the IP layer to select a local IP address before sending the (first) SYN (MUST-44). See the function GET_SRCADDR() in Section 3.4 of RFC 1122.

At all other times, a previous segment has either been sent or received on this connection, and TCP MUST use the same local address is used that was used in those previous segments (MUST-45).

A TCP implementation MUST reject as an error a local OPEN call for an invalid remote IP address (e.g., a broadcast or multicast address) (MUST-46).

Send

Format: SEND (local connection name, buffer address, byte count, PUSH flag (optional), URGENT flag [,timeout])

This call causes the data contained in the indicated user buffer to be sent on the indicated connection. If the connection has not been opened, the SEND is considered an error. Some implementations may allow users to SEND first; in which case, an automatic OPEN would be done. For example, this might be one way for application data to be included in SYN segments. If the calling process is not authorized to use this connection, an error is returned.

A TCP MAY implement PUSH flags on SEND calls (MAY-15). If PUSH flags are not implemented, then the sending TCP: (1) MUST NOT buffer data indefinitely (MUST-60), and (2) MUST set the PSH bit in the last buffered segment (i.e., when there is no more queued data to be sent) (MUST-61). The remaining description below assumes the PUSH flag is supported on SEND calls.

If the PUSH flag is set, the application intends the data to be transmitted promptly to the receiver, and the PUSH bit will be set in the last TCP segment created from the buffer. When an application issues a series of SEND calls without setting the PUSH flag, the TCP MAY aggregate the data internally without sending it (MAY-16).

The PSH bit is not a record marker and is independent of segment boundaries. The transmitter SHOULD collapse successive bits when it packetizes data, to send the largest possible segment (SHLD-27).

If the PUSH flag is not set, the data may be combined with data from subsequent SENDs for transmission efficiency. Note that when the Nagle algorithm is in use, TCP may buffer the data before sending, without regard to the PUSH flag (see Section 3.7.4).

An application program is logically required to set the PUSH flag in a SEND call whenever it needs to force delivery of the data to avoid a communication deadlock. However, a TCP SHOULD send a maximum-sized segment whenever possible (SHLD-28), to improve performance (see Section 3.8.6.2.1).

New applications SHOULD NOT set the URGENT flag [29] due to implementation differences and middlebox issues (SHLD-13).

If the URGENT flag is set, segments sent to the destination TCP will have the urgent pointer set. The receiving TCP will signal the urgent condition to the receiving process if the urgent pointer indicates that data preceding the urgent pointer has not been consumed by the receiving process. The purpose of urgent is to stimulate the receiver to process the urgent data and to indicate to the receiver when all the currently known urgent data has been received. The number of times the sending user's TCP signals urgent will not necessarily be equal to the number of times the receiving user will be notified of the presence of urgent data.

If no foreign socket was specified in the OPEN, but the connection is established (e.g., because a LISTENing connection has become specific due to a foreign segment arriving for the local socket), then the designated buffer is sent to the implied foreign socket. Users who make use of OPEN with an unspecified foreign socket can make use of SEND without ever explicitly knowing the foreign socket address.

However, if a SEND is attempted before the foreign socket becomes specified, an error will be returned. Users can use the STATUS call to determine the status of the connection. In some implementations the TCP may notify the user when an unspecified socket is bound.

If a timeout is specified, the current user timeout for this connection is changed to the new one.

In the simplest implementation, SEND would not return control to the sending process until either the transmission was complete or the timeout had been exceeded. However, this simple method is both subject to deadlocks (for example, both sides of the connection might try to do SENDs before doing any RECEIVEs) and offers poor performance, so it is not recommended. A more sophisticated implementation would return immediately to allow the process to run concurrently with network I/O, and, furthermore, to allow multiple SENDs to be in progress. Multiple SENDs are served in first come, first served order, so the TCP will queue those it cannot service immediately.

We have implicitly assumed an asynchronous user interface in which a SEND later elicits some kind of SIGNAL or pseudo-interrupt from the serving TCP. An alternative is to return a response immediately. For instance, SENDs might return immediate local acknowledgment, even if the segment sent had not been acknowledged by the distant TCP. We could optimistically assume eventual success. If we are wrong, the connection will close anyway due to the timeout. In implementations of this kind (synchronous), there will still be some asynchronous signals, but these will deal with the connection itself, and not with specific segments or buffers.

In order for the process to distinguish among error or success indications for different SENDs, it might be appropriate for the buffer address to be returned along with the coded response to the SEND request. TCP-to-user signals are discussed below, indicating the information which should be returned to the calling process.

Receive

Format: RECEIVE (local connection name, buffer address, byte count) -> byte count, urgent flag, push flag (optional)

This command allocates a receiving buffer associated with the specified connection. If no OPEN precedes this command or the calling process is not authorized to use this connection, an error is returned.

In the simplest implementation, control would not return to the calling program until either the buffer was filled, or some error occurred, but this scheme is highly subject to deadlocks. A more sophisticated implementation would permit several RECEIVEs to be outstanding at once. These would be filled as segments arrive. This strategy permits increased throughput at the cost of a more elaborate scheme (possibly asynchronous) to notify the calling program that a PUSH has been seen or a buffer filled.

A TCP receiver MAY pass a received PSH flag to the application layer via the PUSH flag in the interface (MAY-17), but it is not required (this was clarified in RFC 1122 section 4.2.2.2). The remainder of text describing the RECEIVE call below assumes that passing the PUSH indication is supported.

If enough data arrive to fill the buffer before a PUSH is seen, the PUSH flag will not be set in the response to the RECEIVE. The buffer will be filled with as much data as it can hold. If a PUSH is seen before the buffer is filled the buffer will be returned partially filled and PUSH indicated.

If there is urgent data the user will have been informed as soon as it arrived via a TCP-to-user signal. The receiving user should thus be in "urgent mode". If the URGENT flag is on, additional urgent data remains. If the URGENT flag is off, this call to RECEIVE has returned all the urgent data, and the user may now leave "urgent mode". Note that data following the urgent pointer (non-urgent data) cannot be delivered to the user in the same buffer with preceding urgent data unless the boundary is clearly marked for the user.

To distinguish among several outstanding RECEIVEs and to take care of the case that a buffer is not completely filled, the return code is accompanied by both a buffer pointer and a byte count indicating the actual length of the data received.

Alternative implementations of RECEIVE might have the TCP allocate buffer storage, or the TCP might share a ring buffer with the user.

Close

Format: CLOSE (local connection name)

This command causes the connection specified to be closed. If the connection is not open or the calling process is not authorized to use this connection, an error is returned. Closing connections is intended to be a graceful operation in the sense that outstanding SENDs will be transmitted (and retransmitted), as flow control permits, until all have been serviced. Thus, it should be acceptable to make several SEND calls, followed by a CLOSE, and expect all the data to be sent to the destination. It should also be clear that users should continue to RECEIVE on CLOSING connections, since the other side may be trying to transmit the last of its data. Thus, CLOSE means "I have no more to send" but does not mean "I will not receive any more." It may happen (if the user level protocol is not well thought out) that the closing side is unable to get rid of all its data before timing out. In this event, CLOSE turns into ABORT, and the closing TCP gives up.

The user may CLOSE the connection at any time on his own initiative, or in response to various prompts from the TCP (e.g., remote close executed, transmission timeout exceeded, destination inaccessible).

Because closing a connection requires communication with the foreign TCP, connections may remain in the closing state for a short time. Attempts to reopen the connection before the TCP replies to the CLOSE command will result in error responses.

Close also implies push function.

Status

Format: STATUS (local connection name) -> status data

This is an implementation dependent user command and could be excluded without adverse effect. Information returned would typically come from the TCB associated with the connection.

This command returns a data block containing the following information:

local socket,
foreign socket,
local connection name,
receive window,
send window,
connection state,
number of buffers awaiting acknowledgment,
number of buffers pending receipt,
urgent state,
DiffServ field value,
security/compartment,
and transmission timeout.

Depending on the state of the connection, or on the implementation itself, some of this information may not be available or meaningful. If the calling process is not authorized to use this connection, an error is returned. This prevents unauthorized processes from gaining information about a connection.

Abort

Format: ABORT (local connection name)

This command causes all pending SENDs and RECEIVES to be aborted, the TCB to be removed, and a special RESET message to be sent to the TCP on the other side of the connection. Depending on the implementation, users may receive abort indications for each outstanding SEND or RECEIVE, or may simply receive an ABORT-acknowledgment.

Flush

Some TCP implementations have included a FLUSH call, which will empty the TCP send queue of any data for which the user has issued SEND calls but which is still to the right of the current send window. That is, it flushes as much queued send data as possible without losing sequence number synchronization. The FLUSH call MAY be implemented (MAY-14).

Asynchronous Reports

There MUST be a mechanism for reporting soft TCP error conditions to the application (MUST-47). Generically, we assume this takes the form of an application-supplied ERROR_REPORT routine that may be upcalled asynchronously from the transport layer:

ERROR_REPORT(local connection name, reason, subreason)

The precise encoding of the reason and subreason parameters is not specified here. However, the conditions that are reported asynchronously to the application MUST include:

* ICMP error message arrived (see Section 3.9.2.2 for description of handling each ICMP message type, since some message types need to be suppressed from generating reports to the application)

* Excessive retransmissions (see Section 3.8.3) (TODO - the MUST here is inconsistent with SHOULD in the section describing excessive retransmissions. Both conflicting bits of text are direct from 1122)

* Urgent pointer advance (see Section 3.8.5) (MUST-32).

However, an application program that does not want to receive such ERROR_REPORT calls SHOULD be able to effectively disable these calls (SHLD-20).

Set Differentiated Services Field (IPv4 TOS or IPv6 Traffic Class)

The application layer MUST be able to specify the Differentiated Services field for segments that are sent on a connection (MUST-48). The Differentiated Services field includes the 6-bit Differentiated Services Code Point (DSCP) value. It is not required, but the application SHOULD be able to change the Differentiated Services field during the connection lifetime (SHLD-21). TCP SHOULD pass the current Differentiated Services field value without change to the IP layer, when it sends segments on the connection (SHLD-22).

The Differentiated Services field will be specified independently in each direction on the connection, so that the receiver application will specify the Differentiated Services field used for ACK segments.

TCP MAY pass the most recently received Differentiated Services field up to the application (MAY-9).

3.9.2. TCP/Lower-Level Interface

The TCP calls on a lower level protocol module to actually send and receive information over a network. The two current standard Internet Protocol (IP) versions layered below TCP are IPv4 [1] and IPv6 [11].

If the lower level protocol is IPv4 it provides arguments for a type of service (used within the Differentiated Services field) and for a time to live. TCP uses the following settings for these parameters:

DiffServ field: The IP header value for the DiffServ field is given by the user. This includes the bits of the DiffServ Code Point (DSCP).

Time to Live (TTL): The TTL value used to send TCP segments MUST be configurable (MUST-49).

Note that RFC 793 specified one minute (60 seconds) as a constant for the TTL, because the assumed maximum segment lifetime was two minutes. This was intended to explicitly ask that a segment be destroyed if it cannot be delivered by the internet system within one minute. RFC 1122 changed this specification to require that the TTL be configurable.

Note that the DiffServ field is permitted to change during a connection (section 4.2.4.2 of RFC 1122). However, the application interface might not support this ability, and the application does not have knowledge about individual TCP segments, so this can only be done on a coarse granularity, at best. This limitation is further discussed in RFC 7657 (sec 5.1, 5.3, and 6) [38]. Generally, an application SHOULD NOT change the DiffServ field value during the course of a connection (SHLD-23).

Any lower level protocol will have to provide the source address, destination address, and protocol fields, and some way to determine the "TCP length", both to provide the functional equivalent service of IP and to be used in the TCP checksum.

When received options are passed up to TCP from the IP layer, TCP MUST ignore options that it does not understand (MUST-50).

A TCP MAY support the Time Stamp (MAY-10) and Record Route (MAY-11) options.

3.9.2.1. Source Routing

If the lower level is IP (or other protocol that provides this feature) and source routing is used, the interface must allow the route information to be communicated. This is especially important so that the source and destination addresses used in the TCP checksum be the originating source and ultimate destination. It is also important to preserve the return route to answer connection requests.

An application MUST be able to specify a source route when it actively opens a TCP connection (MUST-51), and this MUST take precedence over a source route received in a datagram (MUST-52).

When a TCP connection is OPENed passively and a packet arrives with a completed IP Source Route option (containing a return route), TCP MUST save the return route and use it for all segments sent on this connection (MUST-53). If a different source route arrives in a later segment, the later definition SHOULD override the earlier one (SHLD-24).

3.9.2.2. ICMP Messages

TCP MUST act on an ICMP error message passed up from the IP layer, directing it to the connection that created the error (MUST-54). The necessary demultiplexing information can be found in the IP header contained within the ICMP message.

This applies to ICMPv6 in addition to IPv4 ICMP.

[23] contains discussion of specific ICMP and ICMPv6 messages classified as either "soft" or "hard" errors that may bear different responses. Treatment for classes of ICMP messages is described below:

Source Quench
  TCP MUST silently discard any received ICMP Source Quench messages (MUST-55). See [10] for discussion.

Soft Errors
  For ICMP these include: Destination Unreachable -- codes 0, 1, 5, Time Exceeded -- codes 0, 1, and Parameter Problem.
  For ICMPv6 these include: Destination Unreachable -- codes 0 and 3, Time Exceeded -- codes 0, 1, and Parameter Problem -- codes 0, 1, 2

Since these Unreachable messages indicate soft error conditions, TCP MUST NOT abort the connection (MUST-56), and it SHOULD make the information available to the application (SHLD-25).

Hard Errors
For ICMP these include Destination Unreachable -- codes 2-4">
These are hard error conditions, so TCP SHOULD abort the connection (SHLD-26). [23] notes that some implementations do not abort connections when an ICMP hard error is received for a connection that is in any of the synchronized states.

Note that [23] section 4 describes widespread implementation behavior that treats soft errors as hard errors during connection establishment.

3.9.2.3. Remote Address Validation

RFC 1122 requires addresses to be validated in incoming SYN packets:

> An incoming SYN with an invalid source address MUST be ignored either by TCP or by the IP layer (MUST-63) (see Section 3.2.1.3 of [14]).
>
> A TCP implementation MUST silently discard an incoming SYN segment that is addressed to a broadcast or multicast address (MUST-57).

This prevents connection state and replies from being erroneously generated, and implementers should note that this guidance is applicable to all incoming segments, not just SYNs, as specifically indicated in RFC 1122.

3.10. Event Processing

The processing depicted in this section is an example of one possible implementation. Other implementations may have slightly different processing sequences, but they should differ from those in this section only in detail, not in substance.

The activity of the TCP can be characterized as responding to events. The events that occur can be cast into three categories: user calls, arriving segments, and timeouts. This section describes the processing the TCP does in response to each of the events. In many cases the processing required depends on the state of the connection.

Events that occur:

User Calls

OPEN
SEND
RECEIVE
CLOSE
ABORT
STATUS

Arriving Segments

SEGMENT ARRIVES

Timeouts

USER TIMEOUT
RETRANSMISSION TIMEOUT
TIME-WAIT TIMEOUT

The model of the TCP/user interface is that user commands receive an immediate return and possibly a delayed response via an event or pseudo interrupt. In the following descriptions, the term "signal" means cause a delayed response.

Error responses are given as character strings. For example, user commands referencing connections that do not exist receive "error: connection not open".

Please note in the following that all arithmetic on sequence numbers, acknowledgment numbers, windows, et cetera, is modulo 2**32 the size of the sequence number space. Also note that "=<" means less than or equal to (modulo 2**32).

A natural way to think about processing incoming segments is to imagine that they are first tested for proper sequence number (i.e., that their contents lie in the range of the expected "receive window" in the sequence number space) and then that they are generally queued and processed in sequence number order.

When a segment overlaps other already received segments we reconstruct the segment to contain just the new data, and adjust the header fields to be consistent.

Note that if no state change is mentioned the TCP stays in the same state.

OPEN Call

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

Create a new transmission control block (TCB) to hold connection state information. Fill in local socket identifier, foreign socket, DiffServ field, security/compartment, and user timeout information. Note that some parts of the foreign socket may be unspecified in a passive OPEN and are to be filled in by the parameters of the incoming SYN segment. Verify the security and DiffServ value requested are allowed for this user, if not return "error: precedence not allowed" or "error: security/compartment not allowed." If passive enter the LISTEN state and return. If active and the foreign socket is unspecified, return "error: foreign socket unspecified"; if active and the foreign socket is specified, issue a SYN segment. An initial send sequence number (ISS) is selected. A SYN segment of the form <SEQ=ISS><CTL=SYN> is sent. Set SND.UNA to ISS, SND.NXT to ISS+1, enter SYN-SENT state, and return.

If the caller does not have access to the local socket specified, return "error: connection illegal for this process". If there is no room to create a new connection, return "error: insufficient resources".

LISTEN STATE

If active and the foreign socket is specified, then change the connection from passive to active, select an ISS. Send a SYN segment, set SND.UNA to ISS, SND.NXT to ISS+1. Enter SYN-SENT state. Data associated with SEND may be sent with SYN segment or queued for transmission after entering ESTABLISHED state. The urgent bit if requested in the command must be sent with the data segments sent as a result of this command. If there is no room to queue the request, respond with "error: insufficient resources". If Foreign socket was not specified, then return "error: foreign socket unspecified".

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

Return "error: connection already exists".

SEND Call

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

If the user does not have access to such a connection, then return "error: connection illegal for this process".

Otherwise, return "error: connection does not exist".

LISTEN STATE

If the foreign socket is specified, then change the connection from passive to active, select an ISS. Send a SYN segment, set SND.UNA to ISS, SND.NXT to ISS+1. Enter SYN-SENT state. Data associated with SEND may be sent with SYN segment or queued for transmission after entering ESTABLISHED state. The urgent bit if requested in the command must be sent with the data segments sent as a result of this command. If there is no room to queue the request, respond with "error: insufficient resources". If Foreign socket was not specified, then return "error: foreign socket unspecified".

SYN-SENT STATE
SYN-RECEIVED STATE

Queue the data for transmission after entering ESTABLISHED state. If no space to queue, respond with "error: insufficient resources".

ESTABLISHED STATE
CLOSE-WAIT STATE

Segmentize the buffer and send it with a piggybacked acknowledgment (acknowledgment value = RCV.NXT). If there is insufficient space to remember this buffer, simply return "error: insufficient resources".

If the urgent flag is set, then SND.UP <- SND.NXT and set the urgent pointer in the outgoing segments.

FIN-WAIT-1 STATE
FIN-WAIT-2 STATE
CLOSING STATE
LAST-ACK STATE
TIME-WAIT STATE

Return "error: connection closing" and do not service request.

RECEIVE Call

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

If the user does not have access to such a connection, return "error: connection illegal for this process".

Otherwise return "error: connection does not exist".

LISTEN STATE
SYN-SENT STATE
SYN-RECEIVED STATE

Queue for processing after entering ESTABLISHED state. If there is no room to queue this request, respond with "error: insufficient resources".

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

If insufficient incoming segments are queued to satisfy the request, queue the request. If there is no queue space to remember the RECEIVE, respond with "error: insufficient resources".

Reassemble queued incoming segments into receive buffer and return to user. Mark "push seen" (PUSH) if this is the case.

If RCV.UP is in advance of the data currently being passed to the user notify the user of the presence of urgent data.

When the TCP takes responsibility for delivering data to the user that fact must be communicated to the sender via an acknowledgment. The formation of such an acknowledgment is described below in the discussion of processing an incoming segment.

CLOSE-WAIT STATE

Since the remote side has already sent FIN, RECEIVEs must be satisfied by text already on hand, but not yet delivered to the user. If no text is awaiting delivery, the RECEIVE will get a "error: connection closing" response. Otherwise, any remaining text can be used to satisfy the RECEIVE.

CLOSING STATE
LAST-ACK STATE

TIME-WAIT STATE

Return "error: connection closing".

CLOSE Call

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

If the user does not have access to such a connection, return "error: connection illegal for this process".

Otherwise, return "error: connection does not exist".

LISTEN STATE

Any outstanding RECEIVEs are returned with "error: closing" responses. Delete TCB, enter CLOSED state, and return.

SYN-SENT STATE

Delete the TCB and return "error: closing" responses to any queued SENDs, or RECEIVEs.

SYN-RECEIVED STATE

If no SENDs have been issued and there is no pending data to send, then form a FIN segment and send it, and enter FIN-WAIT-1 state; otherwise queue for processing after entering ESTABLISHED state.

ESTABLISHED STATE

Queue this until all preceding SENDs have been segmentized, then form a FIN segment and send it. In any case, enter FIN-WAIT-1 state.

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

Strictly speaking, this is an error and should receive a "error: connection closing" response. An "ok" response would be acceptable, too, as long as a second FIN is not emitted (the first FIN may be retransmitted though).

CLOSE-WAIT STATE

Queue this request until all preceding SENDs have been segmentized; then send a FIN segment, enter LAST-ACK state.

CLOSING STATE
LAST-ACK STATE
TIME-WAIT STATE

Respond with "error: connection closing".

ABORT Call

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

If the user should not have access to such a connection, return "error: connection illegal for this process".

Otherwise return "error: connection does not exist".

LISTEN STATE

Any outstanding RECEIVEs should be returned with "error: connection reset" responses. Delete TCB, enter CLOSED state, and return.

SYN-SENT STATE

All queued SENDs and RECEIVEs should be given "connection reset" notification, delete the TCB, enter CLOSED state, and return.

SYN-RECEIVED STATE
ESTABLISHED STATE
FIN-WAIT-1 STATE
FIN-WAIT-2 STATE
CLOSE-WAIT STATE

Send a reset segment:

<SEQ=SND.NXT><CTL=RST>

All queued SENDs and RECEIVEs should be given "connection reset" notification; all segments queued for transmission (except for the RST formed above) or retransmission should be flushed, delete the TCB, enter CLOSED state, and return.

CLOSING STATE LAST-ACK STATE TIME-WAIT STATE

Respond with "ok" and delete the TCB, enter CLOSED state, and return.

STATUS Call

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

If the user should not have access to such a connection, return "error: connection illegal for this process".

Otherwise return "error: connection does not exist".

LISTEN STATE

Return "state = LISTEN", and the TCB pointer.

SYN-SENT STATE

Return "state = SYN-SENT", and the TCB pointer.

SYN-RECEIVED STATE

Return "state = SYN-RECEIVED", and the TCB pointer.

ESTABLISHED STATE

Return "state = ESTABLISHED", and the TCB pointer.

FIN-WAIT-1 STATE

Return "state = FIN-WAIT-1", and the TCB pointer.

FIN-WAIT-2 STATE

Return "state = FIN-WAIT-2", and the TCB pointer.

CLOSE-WAIT STATE

Return "state = CLOSE-WAIT", and the TCB pointer.

CLOSING STATE

Return "state = CLOSING", and the TCB pointer.

LAST-ACK STATE

Return "state = LAST-ACK", and the TCB pointer.

TIME-WAIT STATE

Return "state = TIME-WAIT", and the TCB pointer.

SEGMENT ARRIVES

If the state is CLOSED (i.e., TCB does not exist) then

all data in the incoming segment is discarded. An incoming segment containing a RST is discarded. An incoming segment not containing a RST causes a RST to be sent in response. The acknowledgment and sequence field values are selected to make the reset sequence acceptable to the TCP that sent the offending segment.

If the ACK bit is off, sequence number zero is used,

<SEQ=0><ACK=SEG.SEQ+SEG.LEN><CTL=RST,ACK>

If the ACK bit is on,

<SEQ=SEG.ACK><CTL=RST>

Return.

If the state is LISTEN then

first check for an RST

An incoming RST should be ignored. Return.

second check for an ACK

Any acknowledgment is bad if it arrives on a connection still in the LISTEN state. An acceptable reset segment should be formed for any arriving ACK-bearing segment. The RST should be formatted as follows:

<SEQ=SEG.ACK><CTL=RST>

Return.

third check for a SYN

If the SYN bit is set, check the security. If the security/compartment on the incoming segment does not exactly match the security/compartment in the TCB then send a reset and return.

<SEQ=0><ACK=SEG.SEQ+SEG.LEN><CTL=RST,ACK>

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

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

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

fourth other text or control

Any other control or text-bearing segment (not containing SYN) must have an ACK and thus would be discarded by the ACK processing. An incoming RST segment could not be valid, since it could not have been sent in response to anything sent by this incarnation of the connection. So you are unlikely to get here, but if you do, drop the segment, and return.

If the state is SYN-SENT then

first check the ACK bit

If the ACK bit is set

If SEG.ACK =< ISS, or SEG.ACK > SND.NXT, send a reset (unless the RST bit is set, if so drop the segment and return)

<SEQ=SEG.ACK><CTL=RST>

and discard the segment. Return.

If SND.UNA < SEG.ACK =< SND.NXT then the ACK is acceptable. Some deployed TCP code has used the check SEG.ACK == SND.NXT (using "==" rather than "=<", but this is not appropriate when the stack is capable of sending data on the SYN, because the peer TCP may not accept and acknowledge all of the data on the SYN.

second check the RST bit

If the RST bit is set

A potential blind reset attack is described in RFC 5961 [28], with the mitigation that a TCP implementation SHOULD first check that the sequence number exactly matches RCV.NXT prior to executing the action in the next paragraph.

If the ACK was acceptable then signal the user "error: connection reset", drop the segment, enter CLOSED state, delete TCB, and return. Otherwise (no ACK) drop the segment and return.

third check the security

If the security/compartment in the segment does not exactly match the security/compartment in the TCB, send a reset

If there is an ACK

<SEQ=SEG.ACK><CTL=RST>

Otherwise

<SEQ=0><ACK=SEG.SEQ+SEG.LEN><CTL=RST,ACK>

If a reset was sent, discard the segment and return.

fourth check the SYN bit

This step should be reached only if the ACK is ok, or there is no ACK, and it the segment did not contain a RST.

If the SYN bit is on and the security/compartment is acceptable then, RCV.NXT is set to SEG.SEQ+1, IRS is set to SEG.SEQ. SND.UNA should be advanced to equal SEG.ACK (if there is an ACK), and any segments on the retransmission queue which are thereby acknowledged should be removed.

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

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

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

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

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

and send it. Set the variables:

```
SND.WND <- SEG.WND
SND.WL1 <- SEG.SEQ
SND.WL2 <- SEG.ACK
```

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

Note that it is legal to send and receive application data on SYN segments (this is the "text in the segment" mentioned above. There has been significant misinformation and misunderstanding of this topic historically. Some firewalls and security devices consider this suspicious. However, the capability was used in T/TCP [16] and is used in TCP Fast Open (TFO) [36], so is important for implementations and network devices to permit.

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

Otherwise,

first check sequence number

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

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

In general, the processing of received segments MUST be implemented to aggregate ACK segments whenever possible (MUST-58). For example, if the TCP is processing a series of queued segments, it MUST process them all before sending any ACK segments (MUST-59).

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

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

In implementing sequence number validation as described here, please note Appendix A.2.

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

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

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

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

Note that for the TIME-WAIT state, there is an improved algorithm described in [30] for handling incoming SYN segments, that utilizes timestamps rather than relying on the sequence number check described here. When the improved algorithm is implemented, the logic above is not applicable for incoming SYN segments with timestamp options, received on a connection in the TIME-WAIT state.

In the following it is assumed that the segment is the idealized segment that begins at RCV.NXT and does not exceed the window. One could tailor actual segments to fit this assumption by trimming off any portions that lie outside the window (including SYN and FIN), and only processing further if the segment then begins at RCV.NXT. Segments with higher beginning sequence numbers SHOULD be held for later processing (SHLD-31).

second check the RST bit,

RFC 5961 section 3 describes a potential blind reset attack and optional mitigation approach that SHOULD be implemented. For stacks implementing RFC 5961, the three checks below apply, otherwise processesing for these states is indicated further below.

1) If the RST bit is set and the sequence number is outside the current receive window, silently drop the segment.

2) If the RST bit is set and the sequence number exactly matches the next expected sequence number (RCV.NXT), then TCP MUST reset the connection in the manner prescribed below according to the connection state.

3) If the RST bit is set and the sequence number does not exactly match the next expected sequence value, yet is within the current receive window, TCP MUST send an acknowledgement (challenge ACK):

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

After sending the challenge ACK, TCP MUST drop the unacceptable segment and stop processing the incoming packet further. Note that RFC 5961 and Errata ID 4772 contain additional considerations for ACK throttling in an implementation.

SYN-RECEIVED STATE

If the RST bit is set

If this connection was initiated with a passive OPEN (i.e., came from the LISTEN state), then return this connection to LISTEN state and return. The user need not be informed. If this connection was initiated with an active OPEN (i.e., came from SYN-SENT state) then the connection was refused, signal the user "connection refused". In either case, all segments on the retransmission queue should be removed. And in the active OPEN case, enter the CLOSED state and delete the TCB, and return.

ESTABLISHED
FIN-WAIT-1
FIN-WAIT-2
CLOSE-WAIT

If the RST bit is set then, any outstanding RECEIVEs and SEND should receive "reset" responses. All segment queues should be flushed. Users should also receive an unsolicited general "connection reset" signal. Enter the CLOSED state, delete the TCB, and return.

CLOSING STATE
LAST-ACK STATE
TIME-WAIT

If the RST bit is set then, enter the CLOSED state, delete the TCB, and return.

third check security

SYN-RECEIVED

If the security/compartment in the segment does not exactly match the security/compartment in the TCB then send a reset, and return.

ESTABLISHED
FIN-WAIT-1
FIN-WAIT-2
CLOSE-WAIT
CLOSING
LAST-ACK
TIME-WAIT

If the security/compartment in the segment does not exactly match the security/compartment in the TCB then send a reset, any outstanding RECEIVEs and SEND should receive "reset" responses. All segment queues should be flushed. Users should also receive an unsolicited general "connection reset" signal. Enter the CLOSED state, delete the TCB, and return.

Note this check is placed following the sequence check to prevent a segment from an old connection between these ports with a different security from causing an abort of the current connection.

fourth, check the SYN bit,

SYN-RECEIVED

If the connection was initiated with a passive OPEN, then return this connection to the LISTEN state and return. Otherwise, handle per the directions for synchronized states below.

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

If the SYN bit is set in these synchronized states, it may be either a legitimate new connection attempt (e.g. in the case of TIME-WAIT), an error where the connection should be reset, or the result of an attack attempt, as described in RFC 5961 [28]. For the TIME-WAIT state, new connections can be accepted if the timestamp option is used and meets expectations (per [30]). For all other caess, RFC 5961 provides a mitigation that SHOULD be implemented, though there are alternatives (see Section 6). RFC 5961 recommends that in these synchronized states, if the SYN bit is set, irrespective of the sequence number, TCP MUST send a "challenge ACK" to the remote peer:

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

After sending the acknowledgement, TCP MUST drop the unacceptable segment and stop processing further. Note that RFC 5961 and Errata ID 4772 contain additional ACK throttling notes for an implementation.

For implementations that do not follow RFC 5961, the original RFC 793 behavior follows in this paragraph. If the SYN is in the window it is an error, send a reset, any outstanding RECEIVEs and SEND should receive "reset" responses, all segment queues should be flushed, the user should also receive an unsolicited general "connection reset" signal, enter the CLOSED state, delete the TCB, and return.

If the SYN is not in the window this step would not be reached and an ack would have been sent in the first step (sequence number check).

fifth check the ACK field,

if the ACK bit is off drop the segment and return

if the ACK bit is on

RFC 5961 section 5 describes a potential blind data injection attack, and mitigation that implementations MAY choose to include (MAY-12). TCP stacks that implement RFC 5961 MUST add an input check that the ACK value is acceptable only if it is in the range of ((SND.UNA - MAX.SND.WND) =< SEG.ACK =< SND.NXT). All incoming segments whose ACK value doesn't satisfy the above condition MUST be discarded and an ACK sent back. The new state variable MAX.SND.WND is defined as the largest window that the local sender has ever received from its peer (subject to window scaling) or may be hard-coded to a maximum permissible window value. When the ACK value is acceptable, the processing per-state below applies:

SYN-RECEIVED STATE

If SND.UNA < SEG.ACK =< SND.NXT then enter ESTABLISHED state and continue processing with variables below set to:

SND.WND <- SEG.WND
SND.WL1 <- SEG.SEQ
SND.WL2 <- SEG.ACK

If the segment acknowledgment is not acceptable, form a reset segment,

<SEQ=SEG.ACK><CTL=RST>

and send it.

ESTABLISHED STATE

If SND.UNA < SEG.ACK =< SND.NXT then, set SND.UNA <- SEG.ACK. Any segments on the retransmission queue which are thereby entirely acknowledged are removed. Users should receive positive acknowledgments for buffers which have been SENT and fully acknowledged (i.e., SEND buffer should be returned with "ok" response). If the ACK is a duplicate (SEG.ACK =< SND.UNA), it can be ignored. If the ACK acks something not yet sent (SEG.ACK > SND.NXT) then send an ACK, drop the segment, and return.

If SND.UNA =< SEG.ACK =< SND.NXT, the send window should be updated. If (SND.WL1 < SEG.SEQ or (SND.WL1 = SEG.SEQ and SND.WL2 =< SEG.ACK)), set SND.WND <- SEG.WND, set SND.WL1 <- SEG.SEQ, and set SND.WL2 <- SEG.ACK.

Note that SND.WND is an offset from SND.UNA, that SND.WL1 records the sequence number of the last segment used to update SND.WND, and that SND.WL2 records the acknowledgment number of the last segment used to update SND.WND. The check here prevents using old segments to update the window.

FIN-WAIT-1 STATE

In addition to the processing for the ESTABLISHED state, if our FIN is now acknowledged then enter FIN-WAIT-2 and continue processing in that state.

FIN-WAIT-2 STATE

In addition to the processing for the ESTABLISHED state, if the retransmission queue is empty, the user's CLOSE can be acknowledged ("ok") but do not delete the TCB.

CLOSE-WAIT STATE

Do the same processing as for the ESTABLISHED state.

CLOSING STATE

In addition to the processing for the ESTABLISHED state, if the ACK acknowledges our FIN then enter the TIME-WAIT state, otherwise ignore the segment.

LAST-ACK STATE

The only thing that can arrive in this state is an acknowledgment of our FIN. If our FIN is now acknowledged, delete the TCB, enter the CLOSED state, and return.

TIME-WAIT STATE

The only thing that can arrive in this state is a retransmission of the remote FIN. Acknowledge it, and restart the 2 MSL timeout.

sixth, check the URG bit,

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

If the URG bit is set, RCV.UP <- max(RCV.UP,SEG.UP), and signal the user that the remote side has urgent data if the urgent pointer (RCV.UP) is in advance of the data consumed. If the user has already been signaled (or is still in the "urgent mode") for this continuous sequence of urgent data, do not signal the user again.

CLOSE-WAIT STATE
CLOSING STATE
LAST-ACK STATE
TIME-WAIT

This should not occur, since a FIN has been received from the remote side. Ignore the URG.

seventh, process the segment text,

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

Once in the ESTABLISHED state, it is possible to deliver segment text to user RECEIVE buffers. Text from segments can be moved into buffers until either the buffer is full or the segment is empty. If the segment empties and carries an PUSH flag, then the user is informed, when the buffer is returned, that a PUSH has been received.

When the TCP takes responsibility for delivering the data to the user it must also acknowledge the receipt of the data.

Once the TCP takes responsibility for the data it advances RCV.NXT over the data accepted, and adjusts RCV.WND as appropriate to the current buffer availability. The total of RCV.NXT and RCV.WND should not be reduced.

A TCP MAY send an ACK segment acknowledging RCV.NXT when a valid segment arrives that is in the window but not at the left window edge (MAY-13).

Please note the window management suggestions in Section 3.8.

Send an acknowledgment of the form:

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

This acknowledgment should be piggybacked on a segment being transmitted if possible without incurring undue delay.

CLOSE-WAIT STATE
CLOSING STATE
LAST-ACK STATE
TIME-WAIT STATE

This should not occur, since a FIN has been received from the remote side. Ignore the segment text.

eighth, check the FIN bit,

Do not process the FIN if the state is CLOSED, LISTEN or SYN-SENT since the SEG.SEQ cannot be validated; drop the segment and return.

If the FIN bit is set, signal the user "connection closing" and return any pending RECEIVEs with same message, advance RCV.NXT over the FIN, and send an acknowledgment for the FIN. Note that FIN implies PUSH for any segment text not yet delivered to the user.

SYN-RECEIVED STATE
ESTABLISHED STATE

Enter the CLOSE-WAIT state.

FIN-WAIT-1 STATE

If our FIN has been ACKed (perhaps in this segment), then enter TIME-WAIT, start the time-wait timer, turn off the other timers; otherwise enter the CLOSING state.

FIN-WAIT-2 STATE

Enter the TIME-WAIT state. Start the time-wait timer, turn off the other timers.

CLOSE-WAIT STATE

Remain in the CLOSE-WAIT state.

CLOSING STATE

Remain in the CLOSING state.

LAST-ACK STATE

Remain in the LAST-ACK state.

TIME-WAIT STATE

Remain in the TIME-WAIT state. Restart the 2 MSL time-wait timeout.

and return.

USER TIMEOUT

USER TIMEOUT

For any state if the user timeout expires, flush all queues, signal the user "error: connection aborted due to user timeout" in general and for any outstanding calls, delete the TCB, enter the CLOSED state and return.

RETRANSMISSION TIMEOUT

For any state if the retransmission timeout expires on a segment in the retransmission queue, send the segment at the front of the retransmission queue again, reinitialize the retransmission timer, and return.

TIME-WAIT TIMEOUT

If the time-wait timeout expires on a connection delete the TCB, enter the CLOSED state and return.

3.11. Glossary

ACK
: A control bit (acknowledge) occupying no sequence space, which indicates that the acknowledgment field of this segment specifies the next sequence number the sender of this segment is expecting to receive, hence acknowledging receipt of all previous sequence numbers.

connection
: A logical communication path identified by a pair of sockets.

datagram
: A message sent in a packet switched computer communications network.

Destination Address
: The destination address, usually the network and host identifiers.

FIN
: A control bit (finis) occupying one sequence number, which indicates that the sender will send no more data or control occupying sequence space.

fragment
: A portion of a logical unit of data, in particular an internet fragment is a portion of an internet datagram.

header
: Control information at the beginning of a message, segment, fragment, packet or block of data.

host
: A computer. In particular a source or destination of messages from the point of view of the communication network.

Identification
: An Internet Protocol field. This identifying value assigned by the sender aids in assembling the fragments of a datagram.

internet address
: A source or destination address specific to the host level.

internet datagram
: The unit of data exchanged between an internet module and the higher level protocol together with the internet header.

internet fragment
A portion of the data of an internet datagram with an internet header.

IP
Internet Protocol. See [1] and [11].

IRS
The Initial Receive Sequence number. The first sequence number used by the sender on a connection.

ISN
The Initial Sequence Number. The first sequence number used on a connection, (either ISS or IRS). Selected in a way that is unique within a given period of time and is unpredictable to attackers.

ISS
The Initial Send Sequence number. The first sequence number used by the sender on a connection.

left sequence
This is the next sequence number to be acknowledged by the data receiving TCP (or the lowest currently unacknowledged sequence number) and is sometimes referred to as the left edge of the send window.

module
An implementation, usually in software, of a protocol or other procedure.

MSL
Maximum Segment Lifetime, the time a TCP segment can exist in the internetwork system. Arbitrarily defined to be 2 minutes.

octet
An eight bit byte.

Options
An Option field may contain several options, and each option may be several octets in length.

packet
A package of data with a header which may or may not be logically complete. More often a physical packaging than a logical packaging of data.

port
The portion of a socket that specifies which logical input or output channel of a process is associated with the data.

process
A program in execution. A source or destination of data from the point of view of the TCP or other host-to-host protocol.

PUSH
A control bit occupying no sequence space, indicating that this segment contains data that must be pushed through to the receiving user.

RCV.NXT
receive next sequence number

RCV.UP
receive urgent pointer

RCV.WND
receive window

receive next sequence number
This is the next sequence number the local TCP is expecting to receive.

receive window
This represents the sequence numbers the local (receiving) TCP is willing to receive. Thus, the local TCP considers that segments overlapping the range RCV.NXT to RCV.NXT + RCV.WND - 1 carry acceptable data or control. Segments containing sequence numbers entirely outside of this range are considered duplicates and discarded.

RST
A control bit (reset), occupying no sequence space, indicating that the receiver should delete the connection without further interaction. The receiver can determine, based on the sequence number and acknowledgment fields of the incoming segment, whether it should honor the reset command or ignore it. In no case does receipt of a segment containing RST give rise to a RST in response.

SEG.ACK
segment acknowledgment

SEG.LEN
segment length

SEG.SEQ
: segment sequence

SEG.UP
: segment urgent pointer field

SEG.WND
: segment window field

segment
: A logical unit of data, in particular a TCP segment is the unit of data transfered between a pair of TCP modules.

segment acknowledgment
: The sequence number in the acknowledgment field of the arriving segment.

segment length
: The amount of sequence number space occupied by a segment, including any controls which occupy sequence space.

segment sequence
: The number in the sequence field of the arriving segment.

send sequence
: This is the next sequence number the local (sending) TCP will use on the connection. It is initially selected from an initial sequence number curve (ISN) and is incremented for each octet of data or sequenced control transmitted.

send window
: This represents the sequence numbers which the remote (receiving) TCP is willing to receive. It is the value of the window field specified in segments from the remote (data receiving) TCP. The range of new sequence numbers which may be emitted by a TCP lies between SND.NXT and SND.UNA + SND.WND - 1. (Retransmissions of sequence numbers between SND.UNA and SND.NXT are expected, of course.)

SND.NXT
: send sequence

SND.UNA
: left sequence

SND.UP
: send urgent pointer

SND.WL1
segment sequence number at last window update

SND.WL2
segment acknowledgment number at last window update

SND.WND
send window

socket (or socket number)
An address which specifically includes a port identifier, that is, the concatenation of an Internet Address with a TCP port.

Source Address
The source address, usually the network and host identifiers.

SYN
A control bit in the incoming segment, occupying one sequence number, used at the initiation of a connection, to indicate where the sequence numbering will start.

TCB
Transmission control block, the data structure that records the state of a connection.

TCP
Transmission Control Protocol: A host-to-host protocol for reliable communication in internetwork environments.

TOS
Type of Service, an obsoleted IPv4 field. The same header bits currently are used for the Differentiated Services field [5] containing the Differentiated Services Code Point (DSCP) value and two unused bits.

Type of Service
An Internet Protocol field which indicates the type of service for this internet fragment.

URG
A control bit (urgent), occupying no sequence space, used to indicate that the receiving user should be notified to do urgent processing as long as there is data to be consumed with sequence numbers less than the value indicated in the urgent pointer.

urgent pointer

A control field meaningful only when the URG bit is on. This field communicates the value of the urgent pointer which indicates the data octet associated with the sending user's urgent call.

4. Changes from RFC 793

This document obsoletes RFC 793 as well as RFC 6093 and 6528, which updated 793. In all cases, only the normative protocol specification and requirements have been incorporated into this document, and some informational text with background and rationale may not have been carried in. The informational content of those documents is still valuable in learning about and understanding TCP, and they are valid Informational references, even though their normative content has been incorporated into this document.

The main body of this document was adapted from RFC 793's Section 3, titled "FUNCTIONAL SPECIFICATION", with an attempt to keep formatting and layout as close as possible.

The collection of applicable RFC Errata that have been reported and either accepted or held for an update to RFC 793 were incorporated (Errata IDs: 573, 574, 700, 701, 1283, 1561, 1562, 1564, 1565, 1571, 1572, 2296, 2297, 2298, 2748, 2749, 2934, 3213, 3300, 3301). Some errata were not applicable due to other changes (Errata IDs: 572, 575, 1569, 3305, 3602).

Changes to the specification of the Urgent Pointer described in RFC 1122 and 6093 were incorporated. See RFC 6093 for detailed discussion of why these changes were necessary.

The discussion of the RTO from RFC 793 was updated to refer to RFC 6298. The RFC 1122 text on the RTO originally replaced the 793 text, however, RFC 2988 should have updated 1122, and has subsequently been obsoleted by 6298.

RFC 1122 contains a collection of other changes and clarifications to RFC 793. The normative items impacting the protocol have been incorporated here, though some historically useful implementation advice and informative discussion from RFC 1122 is not included here.

RFC 1122 contains more than just TCP requirements, so this document can't obsolete RFC 1122 entirely. It is only marked as "updating" 1122, however, it should be understood to effectively obsolete all of the RFC 1122 material on TCP.

The more secure Initial Sequence Number generation algorithm from RFC 6528 was incorporated. See RFC 6528 for discussion of the attacks that this mitigates, as well as advice on selecting PRF algorithms and managing secret key data.

A note based on RFC 6429 was added to explicitly clarify that system resource mangement concerns allow connection resources to be reclaimed. RFC 6429 is obsoleted in the sense that this clarification has been reflected in this update to the base TCP specification now.

RFC EDITOR'S NOTE: the content below is for detailed change tracking and planning, and not to be included with the final revision of the document.

This document started as draft-eddy-rfc793bis-00, that was merely a proposal and rough plan for updating RFC 793.

The -01 revision of this draft-eddy-rfc793bis incorporates the content of RFC 793 Section 3 titled "FUNCTIONAL SPECIFICATION". Other content from RFC 793 has not been incorporated. The -01 revision of this document makes some minor formatting changes to the RFC 793 content in order to convert the content into XML2RFC format and account for left-out parts of RFC 793. For instance, figure numbering differs and some indentation is not exactly the same.

The -02 revision of draft-eddy-rfc793bis incorporates errata that have been verified:

Errata ID 573: Reported by Bob Braden (note: This errata basically is just a reminder that RFC 1122 updates 793. Some of the associated changes are left pending to a separate revision that incorporates 1122. Bob's mention of PUSH in 793 section 2.8 was not applicable here because that section was not part of the "functional specification". Also the 1122 text on the retransmission timeout also has been updated by subsequent RFCs, so the change here deviates from Bob's suggestion to apply the 1122 text.)
Errata ID 574: Reported by Yin Shuming
Errata ID 700: Reported by Yin Shuming
Errata ID 701: Reported by Yin Shuming
Errata ID 1283: Reported by Pei-chun Cheng
Errata ID 1561: Reported by Constantin Hagemeier
Errata ID 1562: Reported by Constantin Hagemeier
Errata ID 1564: Reported by Constantin Hagemeier
Errata ID 1565: Reported by Constantin Hagemeier
Errata ID 1571: Reported by Constantin Hagemeier
Errata ID 1572: Reported by Constantin Hagemeier
Errata ID 2296: Reported by Vishwas Manral
Errata ID 2297: Reported by Vishwas Manral

Errata ID 2298: Reported by Vishwas Manral
Errata ID 2748: Reported by Mykyta Yevstifeyev
Errata ID 2749: Reported by Mykyta Yevstifeyev
Errata ID 2934: Reported by Constantin Hagemeier
Errata ID 3213: Reported by EugnJun Yi
Errata ID 3300: Reported by Botong Huang
Errata ID 3301: Reported by Botong Huang
Errata ID 3305: Reported by Botong Huang
Note: Some verified errata were not used in this update, as they relate to sections of RFC 793 elided from this document. These include Errata ID 572, 575, and 1569.
Note: Errata ID 3602 was not applied in this revision as it is duplicative of the 1122 corrections.

Not related to RFC 793 content, this revision also makes small tweaks to the introductory text, fixes indentation of the pseudoheader diagram, and notes that the Security Considerations should also include privacy, when this section is written.

The -03 revision of draft-eddy-rfc793bis revises all discussion of the urgent pointer in order to comply with RFC 6093, 1122, and 1011. Since 1122 held requirements on the urgent pointer, the full list of requirements was brought into an appendix of this document, so that it can be updated as-needed.

The -04 revision of draft-eddy-rfc793bis includes the ISN generation changes from RFC 6528.

The -05 revision of draft-eddy-rfc793bis incorporates MSS requirements and definitions from RFC 879, 1122, and 6691, as well as option-handling requirements from RFC 1122.

The -00 revision of draft-ietf-tcpm-rfc793bis incorporates several additional clarifications and updates to the section on segmentation, many of which are based on feedback from Joe Touch improving from the initial text on this in the previous revision.

The -01 revision incorporates the change to Reserved bits due to ECN, as well as many other changes that come from RFC 1122.

The -02 revision has small formating modifications in order to address xml2rfc warnings about long lines. It was a quick update to avoid document expiration. TCPM working group discussion in 2015 also indicated that that we should not try to add sections on implementation advice or similar non-normative information.

The -03 revision incorporates more content from RFC 1122: Passive OPEN Calls, Time-To-Live, Multihoming, IP Options, ICMP messages, Data Communications, When to Send Data, When to Send a Window Update, Managing the Window, Probing Zero Windows, When to Send an ACK Segment. The section on data communications was re-organized into clearer subsections (previously headings were embedded in the 793 text), and windows management advice from 793 was removed (as reviewed by TCPM working group) in favor of the 1122 additions on SWS, ZWP, and related topics.

The -04 revision includes reference to RFC 6429 on the ZWP condition, RFC1122 material on TCP Connection Failures, TCP Keep-Alives, Acknowledging Queued Segments, and Remote Address Validation. RTO computation is referenced from RFC 6298 rather than RFC 1122.

The -05 revision includes the requirement to implement TCP congestion control with recommendation to implemente ECN, the RFC 6633 update to 1122, which changed the requirement on responding to source quench ICMP messages, and discussion of ICMP (and ICMPv6) soft and hard errors per RFC 5461 (ICMPv6 handling for TCP doesn't seem to be mentioned elsewhere in standards track).

The -06 revision includes an appendix on "Other Implementation Notes" to capture widely-deployed fundamental features that are not contained in the RFC series yet. It also added mention of RFC 6994 and the IANA TCP parameters registry as a reference. It includes references to RFC 5961 in appropriate places. The references to TOS were changed to DiffServ field, based on reflecting RFC 2474 as well as the IPv6 presence of traffic class (carrying DiffServ field) rather than TOS.

The -07 revision includes reference to RFC 6191, updated security considerations, discussion of additional implementation considerations, and clarification of data on the SYN.

The -08 revision includes changes based on:

describing treatment of reserved bits (following TCPM mailing list thread from July 2014 on "793bis item - reserved bit behavior"
addition a brief TCP key concepts section to make up for not including the outdated section 2 of RFC 793
changed "TCP" to "host" to resolve conflict between 1122 wording on whether TCP or the network layer chooses an address when multihomed
fixed/updated definition of options in glossary
moved note on aggregating ACKs from 1122 to a more appropriate location
resolved notes on IP precedence and security/compartment
added implementation note on sequence number validation
added note that PUSH does not apply when Nagle is active

added 1122 content on asynchronous reports to replace 793 section on TCP to user messages

The -09 revision fixes section numbering problems.

The -10 revision includes additions to the security considerations based on comments from Joe Touch, and suggested edits on RST/FIN notification, RFC 2525 reference, and other edits suggested by Yuchung Cheng, as well as modifications to DiffServ text from Yuchung Cheng and Gorry Fairhurst.

The -11 revision includes a start at identifying all of the requirements text and referencing each instance in the common table at the end of the document.

The -12 revision completes the requirement language indexing started in -11 and adds necessary description of the PUSH functionality that was missing.

The -13 revision contains only changes in the inline editor notes.

The -14 revision includes updates with regard to several comments from the mailing list, including editorial fixes, adding IANA considerations for the header flags, improving figure title placement, and breaking up the "Terminology" section into more appropriately titled subsections.

Some other suggested changes that will not be incorporated in this 793 update unless TCPM consensus changes with regard to scope are:

1. Tony Sabatini's suggestion for describing DO field
2. Per discussion with Joe Touch (TAPS list, 6/20/2015), the description of the API could be revisited

Early in the process of updating RFC 793, Scott Brim mentioned that this should include a PERPASS/privacy review. This may be something for the chairs or AD to request during WGLC or IETF LC.

5. IANA Considerations

In the "Transmission Control Protocol (TCP) Header Flags" registry, IANA is asked to assign values indicated below. RFC 3168 originally created this registry, but only populated it with the new bits defined in RFC 3168, not these earlier bits that had been described in RFC 793 and earlier documents.

TCP Header Flags

Bit	Name	Reference
10	Urgent Pointer field significant (URG)	(this document)
11	Acknowledgment field significant (ACK)	(this document)
12	Push Function (PSH)	(this document)
13	Reset the connection (RST)	(this document)
14	Synchronize sequence numbers (SYN)	(this document)
15	No more data from sender (FIN)	(this document)

6. Security and Privacy Considerations

The TCP design includes only rudimentary security features that improve the robustness and reliability of connections and application data transfer, but there are no built-in cryptographic capabilities to support any form of privacy, authentication, or other typical security functions. Non-cryptographic enhancements (e.g. [28]) have been developed to improve robustness of TCP connections to particular types of attacks, but the applicability and protections of non-cryptographic enhancements are limited (e.g. see section 1.1 of [28]). Applications typically utilize lower-layer (e.g. IPsec) and upper-layer (e.g. TLS) protocols to provide security and privacy for TCP connections and application data carried in TCP. Methods based on TCP options have been developed as well, to support some security capabilities.

In order to fully protect TCP connections (including their control flags) IPsec or the TCP Authentication Option (TCP-AO) [27] are the only current effective methods. Other methods discussed in this section may protect the payload, but either only a subset of the fields (e.g. tcpcrypt) or none at all (e.g. TLS). Other security features that have been added to TCP (e.g. ISN generation, sequence number checks, etc.) are only capable of partially hindering attacks.

Applications using long-lived TCP flows have been vulnerable to attacks that exploit the processing of control flags described in earlier TCP specifications [21]. TCP-MD5 was a commonly implemented TCP option to support authentication for some of these connections, but had flaws and is now deprecated. TCP-AO provides a capability to protect long-lived TCP connections from attacks, and has superior properties to TCP-MD5. It does not provide any privacy for application data, nor for the TCP headers.

The "tcpcrypt" [45]Experimental extension to TCP provides the ability to cryptographically protect connection data. Metadata aspects of the TCP flow are still visible, but the application stream is well-protected. Within the TCP header, only the urgent pointer and FIN flag are protected through tcpcrypt.

The TCP Roadmap [37] includes notes about several RFCs related to TCP security. Many of the enhancements provided by these RFCs have been integrated into the present document, including ISN generation, mitigating blind in-window attacks, and improving handling of soft errors and ICMP packets. These are all discussed in greater detail in the referenced RFCs that originally described the changes needed to earlier TCP specifications. Additionally, see RFC 6093 [29] for discussion of security considerations related to the urgent pointer field, that has been deprecated.

Since TCP is often used for bulk transfer flows, some attacks are possible that abuse the TCP congestion control logic. An example is "ACK-division" attacks. Updates that have been made to the TCP congestion control specifications include mechanisms like Appropriate Byte Counting (ABC) that act as mitigations to these attacks.

Other attacks are focused on exhausting the resources of a TCP server. Examples include SYN flooding [20] or wasting resources on non-progressing connections [31]. Operating systems commonly implement mitigations for these attacks. Some common defenses also utilize proxies, stateful firewalls, and other technologies outside of the end-host TCP implementation.

7. Acknowledgements

This document is largely a revision of RFC 793, which Jon Postel was the editor of. Due to his excellent work, it was able to last for three decades before we felt the need to revise it.

Andre Oppermann was a contributor and helped to edit the first revision of this document.

We are thankful for the assistance of the IETF TCPM working group chairs, over the course of work on this document:

Michael Scharf
Yoshifumi Nishida
Pasi Sarolahti
Michael Tuexen

During early discussion of this work on the TCPM mailing list, and at the IETF 88 meeting in Vancouver, and following adoption by the TCPM working group, helpful comments, critiques, and reviews were received from (listed alphabetically): David Borman, Mohamed Boucadair, Yuchung Cheng, Martin Duke, Ted Faber, Rodney Grimes, Kevin Lahey, Kevin Mason, Matt Mathis, Tommy Pauly, Hagen Paul Pfeifer, Anthony Sabatini, Michael Scharf, Greg Skinner, Joe Touch, Reji Varghese, Tim Wicinski, Lloyd Wood, and Alex Zimmermann. Joe Touch provided additional help in clarifying the description of segment size parameters and PMTUD/PLPMTUD recommendations.

This document includes content from errata that were reported by (listed chronologically): Yin Shuming, Bob Braden, Morris M. Keesan, Pei-chun Cheng, Constantin Hagemeier, Vishwas Manral, Mykyta Yevstifeyev, EungJun Yi, Botong Huang.

8. References

8.1. Normative References

[1] Postel, J., "Internet Protocol", STD 5, RFC 791, DOI 10.17487/RFC0791, September 1981, <https://www.rfc-editor.org/info/rfc791>.

[2] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, DOI 10.17487/RFC1191, November 1990, <https://www.rfc-editor.org/info/rfc1191>.

[3] 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>.

[4] 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>.

[5] Nichols, K., Blake, S., Baker, F., and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", RFC 2474, DOI 10.17487/RFC2474, December 1998, <https://www.rfc-editor.org/info/rfc2474>.

[6] Borman, D., Deering, S., and R. Hinden, "IPv6 Jumbograms", RFC 2675, DOI 10.17487/RFC2675, August 1999, <https://www.rfc-editor.org/info/rfc2675>.

[7] Lahey, K., "TCP Problems with Path MTU Discovery", RFC 2923, DOI 10.17487/RFC2923, September 2000, <https://www.rfc-editor.org/info/rfc2923>.

[8] 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>.

[9] 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>.

[10] Gont, F., "Deprecation of ICMP Source Quench Messages", RFC 6633, DOI 10.17487/RFC6633, May 2012, <https://www.rfc-editor.org/info/rfc6633>.

[11] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", STD 86, RFC 8200, DOI 10.17487/RFC8200, July 2017, <https://www.rfc-editor.org/info/rfc8200>.

8.2. Informative References

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

[13] Nagle, J., "Congestion Control in IP/TCP Internetworks", RFC 896, DOI 10.17487/RFC0896, January 1984, <https://www.rfc-editor.org/info/rfc896>.

[14] 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>.

[15] Almquist, P., "Type of Service in the Internet Protocol Suite", RFC 1349, DOI 10.17487/RFC1349, July 1992, <https://www.rfc-editor.org/info/rfc1349>.

[16] Braden, R., "T/TCP -- TCP Extensions for Transactions Functional Specification", RFC 1644, DOI 10.17487/RFC1644, July 1994, <https://www.rfc-editor.org/info/rfc1644>.

[17] Paxson, V., Allman, M., Dawson, S., Fenner, W., Griner, J., Heavens, I., Lahey, K., Semke, J., and B. Volz, "Known TCP Implementation Problems", RFC 2525, DOI 10.17487/RFC2525, March 1999, <https://www.rfc-editor.org/info/rfc2525>.

[18] Xiao, X., Hannan, A., Paxson, V., and E. Crabbe, "TCP Processing of the IPv4 Precedence Field", RFC 2873, DOI 10.17487/RFC2873, June 2000, <https://www.rfc-editor.org/info/rfc2873>.

[19] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, DOI 10.17487/RFC4821, March 2007, <https://www.rfc-editor.org/info/rfc4821>.

[20] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", RFC 4987, DOI 10.17487/RFC4987, August 2007, <https://www.rfc-editor.org/info/rfc4987>.

[21] Touch, J., "Defending TCP Against Spoofing Attacks", RFC 4953, DOI 10.17487/RFC4953, July 2007, <https://www.rfc-editor.org/info/rfc4953>.

[22] Culley, P., Elzur, U., Recio, R., Bailey, S., and J. Carrier, "Marker PDU Aligned Framing for TCP Specification", RFC 5044, DOI 10.17487/RFC5044, October 2007, <https://www.rfc-editor.org/info/rfc5044>.

[23] Gont, F., "TCP's Reaction to Soft Errors", RFC 5461, DOI 10.17487/RFC5461, February 2009, <https://www.rfc-editor.org/info/rfc5461>.

[24] StJohns, M., Atkinson, R., and G. Thomas, "Common Architecture Label IPv6 Security Option (CALIPSO)", RFC 5570, DOI 10.17487/RFC5570, July 2009, <https://www.rfc-editor.org/info/rfc5570>.

[25] 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>.

[26] Sandlund, K., Pelletier, G., and L-E. Jonsson, "The RObust Header Compression (ROHC) Framework", RFC 5795, DOI 10.17487/RFC5795, March 2010, <https://www.rfc-editor.org/info/rfc5795>.

[27] 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>.

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

[29] Gont, F. and A. Yourtchenko, "On the Implementation of the TCP Urgent Mechanism", RFC 6093, DOI 10.17487/RFC6093, January 2011, <https://www.rfc-editor.org/info/rfc6093>.

[30] Gont, F., "Reducing the TIME-WAIT State Using TCP Timestamps", BCP 159, RFC 6191, DOI 10.17487/RFC6191, April 2011, <https://www.rfc-editor.org/info/rfc6191>.

[31] Bashyam, M., Jethanandani, M., and A. Ramaiah, "TCP Sender Clarification for Persist Condition", RFC 6429, DOI 10.17487/RFC6429, December 2011, <https://www.rfc-editor.org/info/rfc6429>.

[32] Gont, F. and S. Bellovin, "Defending against Sequence Number Attacks", RFC 6528, DOI 10.17487/RFC6528, February 2012, <https://www.rfc-editor.org/info/rfc6528>.

[33] Borman, D., "TCP Options and Maximum Segment Size (MSS)", RFC 6691, DOI 10.17487/RFC6691, July 2012, <https://www.rfc-editor.org/info/rfc6691>.

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

[35] 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>.

[36] 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>.

[37] 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>.

[38] Black, D., Ed. and P. Jones, "Differentiated Services (Diffserv) and Real-Time Communication", RFC 7657, DOI 10.17487/RFC7657, November 2015, <https://www.rfc-editor.org/info/rfc7657>.

[39] 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>.

[40] Fairhurst, G., Ed., Trammell, B., Ed., and M. Kuehlewind, Ed., "Services Provided by IETF Transport Protocols and Congestion Control Mechanisms", RFC 8095, DOI 10.17487/RFC8095, March 2017, <https://www.rfc-editor.org/info/rfc8095>.

[41] IANA, "Transmission Control Protocol (TCP) Parameters, https://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml", 2019.

[42] IANA, "Transmission Control Protocol (TCP) Header Flags, https://www.iana.org/assignments/tcp-header-flags/tcp-header-flags.xhtml", 2019.

[43] Gont, F., "Processing of IP Security/Compartment and Precedence Information by TCP", draft-gont-tcpm-tcp-seccomp-prec-00 (work in progress), March 2012.

[44] Gont, F. and D. Borman, "On the Validation of TCP Sequence Numbers", draft-gont-tcpm-tcp-seq-validation-02 (work in progress), March 2015.

[45] Bittau, A., Giffin, D., Handley, M., Mazieres, D., Slack, Q., and E. Smith, "Cryptographic protection of TCP Streams (tcpcrypt)", draft-ietf-tcpinc-tcpcrypt-09 (work in progress), November 2017.

[46] Minshall, G., "A Proposed Modification to Nagle's Algorithm", draft-minshall-nagle-01 (work in progress), June 1999.

[47] Dalal, Y. and C. Sunshine, "Connection Management in Transport Protocols", Computer Networks Vol. 2, No. 6, pp. 454-473, December 1978.

Appendix A. Other Implementation Notes

This section includes additional notes and references on TCP implementation decisions that are currently not a part of the RFC series or included within the TCP standard. These items can be considered by implementers, but there was not yet a consensus to include them in the standard.

A.1. IP Security Compartment and Precedence

RFC 793 requires checking the IP security compartment and precedence on incoming TCP segments for consistency within a connection, and with application requests. Each of these aspects of IP have become outdated, without specific updates to RFC 793. The issues with precedence were fixed by [18] which is Standards Track, and so this present TCP specification includes those changes. However, the state of IP security options that may be used by MLS systems is not as clean.

Implementers of MLS systems that use IP security options (e.g. IPSO, CIPSO, or CALIPSO) should implement any additional logic appropriate for their requirements.

Reseting connections when incoming packets do not meet expected security compartment or precedence expectations has been recognized as a possible attack vector [43], and there has been discussion about ammending the TCP specification to prevent connections from being aborted due to non-matching IP security compartment and DiffServ codepoint values.

A.2. Sequence Number Validation

There are cases where the TCP sequence number validation rules can prevent ACK fields from being processed. This can result in connection issues, as described in [44], which includes descriptions of potential problems in conditions of simultaneous open, self-connects, simultaneous close, and simultaneous window probes. The document also describes potential changes to the TCP specification to mitigate the issue by expanding the acceptable sequence numbers.

In Internet usage of TCP, these conditions are rarely occuring. Common operating systems include different alternative mitigations, and the standard has not been updated yet to codify one of them, but implementers should consider the problems described in [44].

A.3. Nagle Modification

In common operating systems, both the Nagle algorithm and delayed acknowledgements are implemented and enabled by default. TCP is used by many applications that have a request-response style of communication, where the combination of the Nagle algorithm and delayed acknowledgements can result in poor application performance. A modification to the Nagle algorithm is described in [46] that improves the situation for these applications.

This modification is implemented in some common operating systems, and does not impact TCP interoperability. Additionally, many applications simply disable Nagle, since this is generally supported by a socket option. The TCP standard has not been updated to include this Nagle modification, but implementers may find it beneficial to consider.

A.4. Low Water Mark

TODO - mention the low watermark function that is in Linux - suggested by Michael Welzl

SO_SNDLOWAT and SO_RCVLOWAT would be potential enhancements to the abstract TCP API

TCP_NOTSENT_LOWAT is what Michael is talking about, that helps a sending TCP application to help avoid creating large amounts of buffered data (and corresponding latency). This is useful for applications that are multiplexing data from multiple upper level streams onto a connection, especially when streams may be a mix of interactive/realtime and bulk data transfer.

Appendix B. TCP Requirement Summary

This section is adapted from RFC 1122.

Note that there is no requirement related to PLPMTUD in this list, but that PLPMTUD is recommended.

FEATURE	ReqID	MUST	SHOULD	MAY	SHOULD NOT	MUST NOT	Footnote
Push flag							
Aggregate or queue un-pushed data	MAY-16			x			
Sender collapse successive PSH flags	SHLD-27		x				
SEND call can specify PUSH	MAY-15			x			
If cannot: sender buffer indefinitely	MUST-60					x	
If cannot: PSH last segment	MUST-61	x					

Notify receiving ALP of PSH	MAY-17			x			1
Send max size segment when possible	SHLD-28		x				
Window							
Treat as unsigned number	MUST-1	x					
Handle as 32-bit number	REC-1		x				
Shrink window from right	SHLD-14				x		
- Send new data when window shrinks	SHLD-15				x		
- Retransmit old unacked data within window	SHLD-16		x				
- Time out conn for data past right edge	SHLD-17				x		
Robust against shrinking window	MUST-34	x					
Receiver’s window closed indefinitely	MAY-8			x			
Use standard probing logic	MUST-35	x					
Sender probe zero window	MUST-36	x					
First probe after RTO	SHLD-29		x				
Exponential backoff	SHLD-30		x				
Allow window stay zero indefinitely	MUST-37	x					
Retransmit old data beyond SND.UNA+SND.WND	MAY-7			x			
Urgent Data							
Include support for urgent pointer	MUST-30	x					
Pointer indicates first non-urgent octet	MUST-62	x					
Arbitrary length urgent data sequence	MUST-31	x					
Inform ALP asynchronously of urgent data	MUST-32	x					1
ALP can learn if/how much urgent data Q’d	MUST-33	x					1
ALP employ the urgent mechanism	SHLD-13				x		
TCP Options							
Support the mandatory option set	MUST-4	x					
Receive TCP option in any segment	MUST-5	x					
Ignore unsupported options	MUST-6	x					
Cope with illegal option length	MUST-7	x					
Implement sending & receiving MSS option	MUST-14	x					
IPv4 Send MSS option unless 536	SHLD-5		x				
IPv6 Send MSS option unless 1220	SHLD-5		x				
Send MSS option always	MAY-3			x			
IPv4 Send-MSS default is 536	MUST-15	x					
IPv6 Send-MSS default is 1220	MUST-15	x					
Calculate effective send seg size	MUST-16	x					
MSS accounts for varying MTU	SHLD-6		x				
TCP Checksums							
Sender compute checksum	MUST-2	x					
Receiver check checksum	MUST-3	x					
ISN Selection							
Include a clock-driven ISN generator component	MUST-8	x					
Secure ISN generator with a PRF component	SHLD-1		x				

PRF computable from outside the host	MUST-9					x	
Opening Connections							
Support simultaneous open attempts	MUST-10	x					
SYN-RECEIVED remembers last state	MUST-11	x					
Passive Open call interfere with others	MUST-41					x	
Function: simultan. LISTENs for same port	MUST-42	x					
Ask IP for src address for SYN if necc.	MUST-44	x					
Otherwise, use local addr of conn.	MUST-45	x					
OPEN to broadcast/multicast IP Address	MUST-46					x	
Silently discard seg to bcast/mcast addr	MUST-57	x					
Closing Connections							
RST can contain data	SHLD-2		x				
Inform application of aborted conn	MUST-12	x					
Half-duplex close connections	MAY-1			x			
Send RST to indicate data lost	SHLD-3		x				
In TIME-WAIT state for 2MSL seconds	MUST-13	x					
Accept SYN from TIME-WAIT state	MAY-2			x			
Use Timestamps to reduce TIME-WAIT	SHLD-4		x				
Retransmissions							
Implement RFC 5681	MUST-19	x					
Retransmit with same IP ident	MAY-4			x			
Karn's algorithm	MUST-18	x					
Generating ACK's:							
Aggregate whenever possible	MUST-58	x					
Queue out-of-order segments	SHLD-31		x				
Process all Q'd before send ACK	MUST-59	x					
Send ACK for out-of-order segment	MAY-13			x			
Delayed ACK's	SHLD-18		x				
Delay < 0.5 seconds	MUST-40	x					
Every 2nd full-sized segment ACK'd	SHLD-19	x					
Receiver SWS-Avoidance Algorithm	MUST-39	x					
Sending data							
Configurable TTL	MUST-49	x					
Sender SWS-Avoidance Algorithm	MUST-38	x					
Nagle algorithm	SHLD-7		x				
Application can disable Nagle algorithm	MUST-17	x					
Connection Failures:							
Negative advice to IP on R1 retxs	MUST-20	x					
Close connection on R2 retxs	MUST-20	x					
ALP can set R2	MUST-21	x					1
Inform ALP of R1<=retxs<R2	SHLD-9		x				1
Recommended value for R1	SHLD-10		x				

Recommended value for R2	SHLD-11		x				
Same mechanism for SYNs	MUST-22	x					
R2 at least 3 minutes for SYN	MUST-23	x					
Send Keep-alive Packets:	MAY-5			x			
- Application can request	MUST-24	x					
- Default is "off"	MUST-25	x					
- Only send if idle for interval	MUST-26	x					
- Interval configurable	MUST-27	x					
- Default at least 2 hrs.	MUST-28	x					
- Tolerant of lost ACK’s	MUST-29	x					
- Send with no data	SHLD-12		x				
- Configurable to send garbage octet	MAY-6			x			
IP Options							
Ignore options TCP doesn’t understand	MUST-50	x					
Time Stamp support	MAY-10			x			
Record Route support	MAY-11			x			
Source Route:							
ALP can specify	MUST-51	x					1
Overrides src rt in datagram	MUST-52	x					
Build return route from src rt	MUST-53	x					
Later src route overrides	SHLD-24		x				
Receiving ICMP Messages from IP	MUST-54	x					
Dest. Unreach (0,1,5) => inform ALP	SHLD-25		x				
Dest. Unreach (0,1,5) => abort conn	MUST-56					x	
Dest. Unreach (2-4) => abort conn	SHLD-26		x				
Source Quench => silent discard	MUST-55	x					
Time Exceeded => tell ALP, don’t abort	MUST-56					x	
Param Problem => tell ALP, don’t abort	MUST-56					x	
Address Validation							
Reject OPEN call to invalid IP address	MUST-46	x					
Reject SYN from invalid IP address	MUST-63	x					
Silently discard SYN to bcast/mcast addr	MUST-57	x					
TCP/ALP Interface Services							
Error Report mechanism	MUST-47	x					
ALP can disable Error Report Routine	SHLD-20		x				
ALP can specify DiffServ field for sending	MUST-48	x					
Passed unchanged to IP	SHLD-22		x				
ALP can change DiffServ field during connection	SHLD-21		x				
ALP generally changing DiffServ during conn.	SHLD-23				x		
Pass received DiffServ field up to ALP	MAY-9			x			
FLUSH call	MAY-14			x			
Optional local IP addr parm. in OPEN	MUST-43	x					

```
 RFC 5961 Support:                                 |        | | | | | |
   Implement data injection protection             | MAY-12 | | |x| | |
                                                   |        | | | | | |
 Explicit Congestion Notification:                 |        | | | | | |
   Support ECN                                     | SHLD-8 | |x| | | |
 --------------------------------------------------|--------|-|-|-|-|-|--
```

FOOTNOTES: (1) "ALP" means Application-Layer program.

Author's Address

Wesley M. Eddy (editor)
MTI Systems
US

Email: wes@mti-systems.com

TCP Extended Data Offset Option
draft-ietf-tcpm-tcp-edo-10.txt

Status of this Memo

Copyright Notice

Abstract

TCP segments include a Data Offset field to indicate space for TCP options but the size of the field can limit the space available for complex options such as SACK and Multipath TCP and can limit the combination of such options supported in a single connection. This document updates RFC 793 with an optional TCP extension to that space to support the use of multiple large options. It also explains why the initial SYN of a connection cannot be extending a single segment.

Table of Contents

1. Introduction

TCP's Data Offset (DO)is a 4-bit field, which indicates the number of 32-bit words of the entire TCP header [RFC793]. This limits the current total header size to 60 bytes, of which the basic header occupies 20, leaving 40 bytes for options. These 40 bytes are increasingly becoming a limitation to the development of advanced capabilities, such as when SACK [RFC2018][RFC6675] is combined with either Multipath TCP [RFC6824], TCP-AO [RFC5925], or TCP Fast Open [RFC7413].

This document specifies the TCP Extended Data Offset (EDO) option, and is independent of (and thus compatible with) IPv4 and IPv6. EDO extends the space available for TCP options, except for the initial SYN and SYN/ACK. This document also explains why the option space of the initial SYN segments cannot be extended as individual segments without severe impact on TCP's initial handshake and the SYN/ACK limitation that results from potential middlebox misbehavior. Multiple other TCP extensions are being considered in the TCPM working group in order to address the case of SYN and SYN/ACK segments [Bo14][Br14][To18]. Some of these other extensions can work in conjunction with EDO (e.g., [To18]).

2. Conventions used in this document

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

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

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

3. Motivation

TCP supports headers with a total length of up to 15 32-bit words, as indicated in the 4-bit Data Offset field [RFC793]. This accounts for a total of 60 bytes, of which the default TCP header fields occupy 20 bytes, leaving 40 bytes for options.

TCP connections already use this option space for a variety of capabilities. These include Maximum Segment Size (MSS) [RFC793], Window Scale (WS) [RFC7323], Timestamp (TS) [RFC7323], Selective Acknowledgement (SACK) [RFC2018][RFC6675], TCP Authentication Option (TCP-AO) [RFC5925], Multipath TCP (MP-TCP)_[RFC6824], and TCP User Timeout [RFC5482]. Some options occur only in a SYN or SYN/ACK (MSS, WS), and others vary in size when used in SYN vs. non-SYN segments.

Each of these options consumes space, where some options consuming as much space as available (SACK) and other desired combinations can easily exceed the currently available space. For example, it is not currently possible to use TCP-AO with both TS and MP-TCP in the same non-SYN segment, i.e., to combine accurate round-trip estimation, authentication, and multipath support in the same connection - even though these options can be negotiated during a SYN exchange (10 for TS, 16 for TCP-AO, and 12 for MP-TCP).

TCP EDO is intended to overcome this limitation for non-SYN segments, as well as to increase the space available for SACK blocks. Further discussion of the impact of EDO and existing options is discussed in Section 6.4. Extending SYN segments is much more complicated, as discussed in Section 8.7.

4. Requirements for Extending TCP’s Data Offset

The primary goal of extending the TCP Data Offset field is to increase the space available for TCP options in all segments except the initial SYN.

An important requirement of any such extension is that it not impact legacy endpoints. Endpoints seeking to use this new option should not incur additional delay or segment exchanges to connect to either new endpoints supporting this option or legacy endpoints without this option. We call this a "backward downgrade" capability.

An additional consideration of this extension is avoiding user data corruption in the presence of popular network devices, including middleboxes. Consideration of middlebox misbehavior can also interfere with extension in the SYN/ACK.

5. The TCP EDO Option

TCP EDO extends the option space for all segments except the initial SYN (i.e., SYN set and ACK not set) and SYN/ACK response. EDO is indicated by the TCP option codepoint of EDO-OPT and has two types: EDO Supported and EDO Extension, as discussed in the following subsections.

5.1. EDO Supported

EDO capability is determined in both directions using a single exchange of the EDO Supported option (Figure 1). When EDO is desired on a given connection, the SYN and SYN/ACK segments include the EDO Supported option, which consists of the two required TCP option fields: Kind and Length. The EDO Supported option is used only in the SYN and SYN/ACK segments and only to confirm support for EDO in subsequent segments.

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

Figure 1 TCP EDO Supported option

An endpoint seeking to enable EDO includes the EDO Supported option in the initial SYN. If receiver of that SYN agrees to use EDO, it responds with the EDO Supported option in the SYN/ACK. The EDO Supported option does not extend the TCP option space.

>> Connections using EDO MUST negotiate its availability during the SYN exchange of the initial three-way handshake.

>> An endpoint confirming and agreeing to EDO use MUST respond with the EDO Supported option in its SYN/ACK.

The SYN/ACK uses only the EDO Supported option (and not the EDO Extension option, below) because it may not yet be safe to extend the option space in the reverse direction due to potential middlebox misbehavior (see Section 7.2). Extension of the SYN and SYN/ACK space is addressed as a separate option (see Section 8.7).

5.2. EDO Extension

When EDO is successfully negotiated, all other segments use the EDO Extension option, of which there are two variants (Figure 2 and Figure 3). Both variants are considered equivalent and either variant can be used in any segment where the EDO Extension option is required. Both variants add a Header_Length field (in network-standard byte order), indicating the length of the entire TCP header in 32-bit words. Figure 3 depicts the longer variant, which includes an additional Segment_Length field, which is identical to the TCP pseudoheader TCP Length field and used to detect when segments have been altered in ways that would interfere with EDO (discussed further in Section 5.3).

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

Figure 2 TCP EDO Extension option - simple variant

```
+--------+--------+--------+--------+
|  Kind  | Length |  Header_Length  |
+--------+--------+--------+--------+
|  Segment_Length |
+--------+--------+
```

Figure 3 TCP EDO Extension option - with segment length verification

>> Once enabled on a connection, all segments in both directions MUST include the EDO Extension option. Segments not needing extension MUST set the EDO Extension option Header Length field equal to the Data Offset length.

>> The EDO Extension option MAY be used only if confirmed when the connection transitions to the ESTABLISHED state, e.g., a client is enabled after receiving the EDO Supported option in the SYN/ACK and the server is enabled after seeing the EDO Extension option in the final ACK of the three-way handshake. If either of those segments lacks the appropriate EDO option, the connection MUST NOT use any EDO options on any other segments.

Internet paths may vary after connection establishment, introducing misbehaving middleboxes (see Section 7.2). Using EDO on all segments in both directions allows this condition to be detected.

>> The EDO Supported option MAY occur in an initial SYN as desired (e.g., as expressed by the user/application) and in the SYN/ACK as confirmation, but MUST NOT be inserted in other segments. If the EDO Supported option is received in other segments, it MUST be silently ignored.

>> If EDO has not been negotiated and agreed, the EDO Extension option MUST be silently ignored on subsequent segments. The EDO Extension option MUST NOT be sent in an initial SYN segment or SYN/ACK, and MUST be silently ignored and not acknowledged if so received.

>> If EDO has been negotiated, any subsequent segments arriving without the EDO Extension option MUST be silently ignored. Such events MAY be logged as warning errors and logging MUST be rate limited.

When processing a segment, EDO needs to be visible within the area indicated by the Data Offset field, so that processing can use the EDO Header_length to override the field for that segment.

>> The EDO Extension option MUST occur within the space indicated by the TCP Data Offset.

>> The EDO Extension option indicates the total length of the header. The EDO Header_length field MUST NOT exceed that of the total segment size (i.e., TCP Length).

>> The EDO Header Length MUST be at least as large as the TCP Data Offset field of the segment in which they both appear. When the EDO Header Length equals the Data Offset length, the EDO Extension option is present but it does not extend the option space. When the EDO Header Length is invalid, the TCP segment MUST be silently dropped.

>> The EDO Supported option SHOULD be aligned on a 16-bit boundary and the EDO Extension option SHOULD be aligned on a 32-bit boundary, in both cases for simpler processing.

For example, a segment with only EDO would have a Data Offset of 6 or 7 (depending on the EDO Extension variant used), where EDO would be the first option processed, at which point the EDO Extension option would override the Data Offset and processing would continue until the end of the TCP header as indicated by the EDO Header_length field.

There are cases where it might be useful to process other options before EDO, notably those that determine whether the TCP header is valid, such as authentication, encryption, or alternate checksums. In those cases, the EDO Extension option is preferably the first option after a validation option, and the payload after the Data Offset is treated as user data for the purposes of validation.

>> The EDO Extension option SHOULD occur as early as possible, either first or just after any authentication or encryption, and SHOULD be the last option covered by the Data Offset value.

Other options are generally handled in the same manner as when the EDO option is not active, unless they interact with other options.

One such example is TCP-AO [RFC5925], which optionally ignores the contents of TCP options, so it would need to be aware of EDO to operate correctly when options are excluded from the HMAC calculation.

>> Options that depend on other options, such as TCP-AO [RFC5925] (which may include or exclude options in MAC calculations) MUST also be augmented to interpret the EDO Extension option to operate correctly.

5.3. The two EDO Extension variants

There are two variants of the EDO Extension option; one includes a copy of the TCP segment length, copied from the TCP pseduoheader [RFC793]. The Segment_Length field is added to the longer variant to detect when segments are incorrectly and inappropriately merged by middleboxes or TCP offload processing but without consideration for the additional option space indicated by the EDO Header_Length field. Such effects are described in further detail in Section 7.2.

>> An endpoint MAY use either variant of the EDO Extension option interchangeably.

When the longer, 6-byte variant is used, the Segment_Length field is used to check whether modification of the segment was performed consistent with knowledge of the EDO option. The Segment_Length field will detect any modification of the length of the segment, such as might occur when segments are split or merged, that occurs without also updating the Segment Length field as well. The Segment Length field thus helps endpoints detects devices that merge or split TCP segments without support for EDO. Devices that merge or split TCP segments that support EDO would update the Segment Length field as needed, but would also ensure that the user data is handled separately from the extended option space indicate by EDO.

>> When an endpoint creates a new segment using the 6-byte EDO Extension option, the Segment_Length field is initialized with a copy of the segment length from the TCP pseudoheader.

>> When an endpoint receives a segment using the 6-byte EDO Extension option, it MUST validate the Segment_Length field with the length of the segment as indicated in the TCP pseudoheader. If the segment lengths do not match, the segment MUST be discarded and an error SHOULD be logged in a rate-limited manner.

>> The 6-byte EDO Extension variant SHOULD be used where middlebox or TCP offload support could merge or split TCP segments without

consideration for the EDO option. Because these conditions could occur at either endpoint or along the network path, the 6-byte variant SHOULD be preferred until sufficient evidence for safe use of the 4-byte variant is determined by the community.

The field will not detect other modification of the TCP user data; such modifications would need more complex detection mechanisms, such as checksums or hashes. When these are used, as with IPsec or TCP-AO, the 4-byte variant is sufficient.

>> The 4-byte EDO Extension variant is sufficient when EDO is used in conjunction with other mechanisms that provide integrity protection, such as IPsec or TCP-AO.

6. TCP EDO Interaction with TCP

The following subsections describe how EDO interacts with the TCP specification [RFC793].

6.1. TCP User Interface

The TCP EDO option is enabled on a connection using a mechanism similar to any other per-connection option. In Unix systems, this is typically performed using the 'setsockopt' system call.

>> Implementations can also employ system-wide defaults, however systems SHOULD NOT activate this extension by default to avoid interfering with legacy applications.

>> Due to the potential impacts of legacy middleboxes (discussed in Section 7), a TCP implementation supporting EDO SHOULD log any events within an EDO connection when options that are malformed or show other evidence of tampering arrive. An operating system MAY choose to cache the list of destination endpoints where this has occurred with and block use of EDO on future connections to those endpoints, but this cache MUST be accessible to users/applications on the host. Note that such endpoint assumptions can vary in the presence of load balancers where server implementations vary behind such balancers.

6.2. TCP States and Transitions

TCP EDO does not alter the existing TCP state or state transition mechanisms.

6.3. TCP Segment Processing

TCP EDO alters segment processing during the TCP option processing step. Once detected, the TCP EDO Extension option overrides the TCP Data Offset field for all subsequent option processing. Option processing continues at the next option (if present) after the EDO Extension option.

6.4. Impact on TCP Header Size

The TCP EDO Supported option increases SYN header length by a minimum of 2 bytes, but could increase it by more depending on 32-bit word alignment. Currently popular SYN options total 19 bytes, which leaves more than enough room for the EDO Supported option:

- o SACK permitted (2 bytes in SYN, optionally 2 + 8N bytes after) [RFC2018][RFC6675]
- o Timestamp (10 bytes) [RFC7323]
- o Window scale (3 bytes) [RFC7323]
- o MSS option (4 bytes) [RFC793]

Adding the EDO Supported option would result in a total of 21 bytes of SYN option space.

Subsequent segments would use 10 bytes of option space without any SACK blocks (TS only; WS and MSS are used only in SYN and SYN/ACK) or allow up to 3 SACK blocks before needing to use EDO; with EDO, the number of SACK blocks or additional options would be substantially increased. There are also other options that are emerging in the SYN, including TCP Fast Open, which uses another 6-18 (typically 10) bytes in the SYN/ACK of the first connection and in the SYN of subsequent connections [RFC7413].

TCP EDO can also be negotiated in SYNs with either of the following large options:

- o TCP-AO (authentication) (16 bytes) [RFC5925]
- o Multipath TCP (12 bytes in SYN and SYN/ACK, 20 after) [RFC6824]

Including TCP-AO with TS, WS, SACK increases the SYN option space use to 35 bytes; with Multipath TCP the use is 31 bytes. When Multipath TCP is enabled with the typical options, later segments would require 30 bytes without SACK, thus limiting the SACK option to one block unless EDO is also supported on at least non-SYN segments.

The full combination of the above options (47 bytes for TS, WS, MSS, SACK, TCP-AO, and MPTCP) does not fit in the existing SYN option space and (as noted) that space cannot be extended within a single SYN segment. There has been a proposal to change TS to a 2 byte "TS permitted" signal in the initial SYN, provided it can be safely enabled during the connection later or might be avoided completely [Ni15]. Even using "TS-permitted", the total space is still too large to support in the initial SYN without SYN option space extension [Bo14][Br14][To18].

The EDO Extension option has negligible impact on other headers, because it can either come first or just after security information, and in either case the additional 4 or 6 bytes are easily accommodated within the TCP Data Offset length. Once the EDO option is processed, the entirety of the remainder of the TCP segment is available for any remaining options.

6.5. Connectionless Resets

A RST may arrive during a currently active connection or may be needed to cleanup old state from an abandoned connection. The latter occurs when a new SYN is sent to an endpoint with matching existing connection state, at which point that endpoint responds with a RST and both ends remove stale information.

The EDO Extension option is mandatory on all TCP segments once negotiated, i.e., except in the SYN and SYN/ACK (which establish support) and the RST. A RST may lack the context to know that EDO is active on a connection.

>> The EDO Extension option MAY occur in a RST when the endpoint has connection state that has negotiated EDO. However, unless the RST is generated by an incoming segment that includes an EDO Extension option, the transmitted RST MUST NOT include the EDO Extension option.

6.6. ICMP Handling

ICMP responses are intended to include the IP and the port fields of TCP and UDP headers of typical TCP/IP and UDP/IP packets [RFC792]. This includes the first 8 data bytes of the original datagram, intended to include the transport port numbers used for connection demultiplexing. Later specifications encourage returning as much of the original payload as possible [RFC1812]. In either case, legacy options or new options in the EDO extension area might or might not be included, and so options are generally not assumed to be part of ICMP processing anyway.

7. Interactions with Middleboxes

Middleboxes are on-path devices that typically examine or modify packets in ways that Internet routers do not [RFC3234]. This includes parsing transport headers and/or rewriting transport segments in ways that may affect EDO.

There are several cases to consider:

- Typical NAT/NAPT devices, which modify only IP address and/or TCP port number fields (with associated TCP checksum updates)

- Middleboxes that try to reconstitute TCP data streams, such as for deep-packet inspection for virus scanning

- Middleboxes that modify known TCP header fields

- Middleboxes that rewrite TCP segments

7.1. Middlebox Coexistence with EDO

Middleboxes can coexist with EDO when they either support EDO or when they ignore its impact on segment structure.

NATs and NAPTs, which rewrite IP address and/or transport port fields, are the most common form of middlebox and are not affected by the EDO option.

Middleboxes that support EDO would be those that correctly parse the EDO option. Such boxes can reconstitute the TCP data stream correctly or can modify header fields and/or rewrite segments without impact to EDO.

Conventional TCP proxies terminate the TCP connection in both directions and thus operate as TCP endpoints, such as when a client-middlebox and middlebox-server each have separate TCP connections. They would support EDO by following the host requirements herein on both connections. The use of EDO on one connection is independent of its use on the other in this case.

7.2. Middlebox Interference with EDO

Middleboxes that do not support EDO cannot coexist with its use when they modify segment boundaries or do not forward unknown (e.g., the EDO) options.

So-called "transparent" rewriting proxies, which inappropriately and incorrectly modify TCP segment boundaries, might mix option information with user data if they did not support EDO. Such devices might also interfere with other TCP options such as TCP-AO. There are three types of such boxes:

- o Those that process received options and transmit sent options separately, i.e., although they rewrite segments, they behave as TCP endpoints in both directions.

- o Those that split segments, taking a received segment and emitting two or more segments with revised headers.

- o Those that join segments, receiving multiple segments and emitting a single segment whose data is the concatenation of the components.

In all three cases, EDO is either treated as independent on different sides of such boxes or not. If independent, EDO would either be correctly terminated in either or both directions or disabled due to lack of SYN/ACK confirmation in either or both directions. Problems would occur only when TCP segments with EDO are combined or split while ignoring the EDO option. In the split case, the key concern is if the split happens within the option extension space or if EDO is silently copied to both segments without copying the corresponding extended option space contents. However, the most comprehensive study of these cases indicates that "although middleboxes do split and coalesce segments, none did so while passing unknown options" [Ho11].

Note that the second and third types of middlebox behaviors listed above may create syndromes similar to TCP transmit and receive hardware offload engines that incorrectly modify segments with unknown options.

Middleboxes that silently remove options that they do not implement have been observed [Ho11]. Such boxes interfere with the use of the EDO Extension option in the SYN and SYN/ACK segments because extended option space would be misinterpreted as user data if the EDO Extension option were removed, and this cannot be avoided. This is one reason that SYN and SYN/ACK extension requires alternate mechanisms (see Section 8.7). It is also the reason for the 6-byte EDO Extension variant (see Section 5.3), which can detect such merging or splitting of segments. Further, if such middleboxes become present on a path they could cause similar misinterpretation on segments exchanged in the ESTABLISHED and subsequent states. As a result, this document requires that the EDO Extension option be avoided on the SYN/ACK and that this option needs to be used on all segments once successfully negotiated and encourages use of the 6-byte EDO Extension variant.

Deep-packet inspection systems that inspect TCP segment payloads or attempt to reconstitute the data stream would incorrectly include option data in the reconstituted user data stream, which might interfere with their operation.

>> It can be important to detect misbehavior that could cause EDO space to be misinterpreted as user data. In such cases, EDO SHOULD be used in conjunction with an integrity protection mechanism. This includes the 6-byte EDO Extension variant or stronger mechanisms such as IPsec, TCP-AO, etc. It is useful to note that such protection only helps non-compliant components and enable avoidance (e.g., disabling EDO), but integrity protection alone cannot correct the misinterpretation of EDO space as user data.

This situation is similar to that of ECN and ICMP support in the Internet. In both cases, endpoints have evolved mechanisms for detecting and robustly operating around "black holes". Very similar algorithms are expected to be applicable for EDO.

8. Comparison to Previous Proposals

EDO is the latest in a long line of attempts to increase TCP option space [Al06][Ed08][Ko04][Ra12][Yo11]. The following is a comparison of these approaches to EDO, based partly on a previous summary [Ra12]. This comparison differs from that summary by using a different set of success criteria.

8.1. EDO Criteria

Our criteria for a successful solution are as follows:

- Zero-cost fallback to legacy endpoints.
- Minimal impact on middlebox compatibility.
- No additional side-effects.

Zero-cost fallback requires that upgraded hosts incur no penalty for attempting to use EDO. This disqualifies dual-stack approaches, because the client might have to delay connection establishment to wait for the preferred connection mode to complete. Note that the impact of legacy endpoints that silently reflect unknown options are not considered, as they are already non-compliant with existing TCP requirements [RFC793].

Minimal impact on middlebox compatibility requires that EDO works through simple NAT and NAPT boxes, which modify IP addresses and ports and recompute IPv4 header and TCP segment checksums. Middleboxes that reject unknown options or that process segments in detail without regard for unknown options are not considered; they process segments as if they were an endpoint but do so in ways that are not compliant with existing TCP requirements (e.g., they should have rejected the initial SYN because of its unknown options rather than silently relaying it).

EDO also attempts to avoid creating side-effects, such as might happen if options were split across multiple TCP segments (which could arrive out of order or be lost) or across different TCP connections (which could fail to share fate through firewalls or NAT/NAPTs).

These requirements are similar to those noted in [Ra12], but EDO groups cases of segment modification beyond address and port - such as rewriting, segment drop, sequence number modification, and option stripping - as already in violation of existing TCP requirements regarding unknown options, and so we do not consider their impact on this new option.

8.2. Summary of Approaches

There are three basic ways in which TCP option space extension has been attempted:

1. Use of a TCP option.

2. Redefinition of the existing TCP header fields.

3. Use of option space in multiple TCP segments (split across multiple segments).

A TCP option is the most direct way to extend the option space and is the basis of EDO. This approach cannot extend the option space of the initial SYN.

Redefining existing TCP header fields can be used to either contain additional options or as a pointer indicating alternate ways to interpret the segment payload. All such redefinitions make it difficult to achieve zero-impact backward compatibility, both with legacy endpoints and middleboxes.

Splitting option space across separate segments can create unintended side-effects, such as increased delay to deal with path latency or loss differences.

The following discusses three of the most notable past attempts to extend the TCP option space: Extended Segments, TCPx2, LO/SLO, and LOIC. [Ra12] suggests a few other approaches, including use of TCP option cookies, reuse/overload of other TCP fields (e.g., the URG pointer), or compressing TCP options. None of these is compatible with legacy endpoints or middleboxes.

8.3. Extended Segments

TCP Extended Segments redefined the meaning of currently unused values of the Data Offset (DO) field [Ko04]. TCP defines DO as indicating the length of the TCP header, including options, in 32-bit words. The default TCP header with no options is 5 such words, so the minimum currently valid DO value is 5 (meaning 40 bytes of option space). This document defines interpretations of values 0-4: DO=0 means 48 bytes of option space, DO=1 means 64, DO=2 means 128, DO=3 means 256, and DO=4 means unlimited (e.g., the entire payload is option space). This variant negotiates the use of this capability by using one of these invalid DO values in the initial SYN.

Use of this variant is not backward-compatible with legacy TCP implementations, whether at the desired endpoint or on middleboxes. The variant also defines a way to initiate the feature on the passive side, e.g., using an invalid DO during the SYN/ACK when the initial SYN had a valid DO. This capability allows either side to initiate use of the feature but is also not backward compatible.

8.4. TCPx2

TCPx2 redefines legacy TCP headers by basically doubling all TCP header fields [Al06]. It relies on a new transport protocol number to indicate its use, defeating backward compatibility with all existing TCP capabilities, including firewalls, NATs/NAPTs, and legacy endpoints and applications.

8.5. LO/SLO

The TCP Long Option (LO, [Ed08]) is very similar to EDO, except that presence of LO results in ignoring the existing Data Offset (DO) field and that LO is required to be the first option. EDO considers the need for other fields to be first and declares that the EDO is the last option as indicated by the DO field value. Like LO, EDO is required in every segment once negotiated.

The TCP Long Option draft also specified the SYN Long Option (SLO) [Ed08]. If SLO is used in the initial SYN and successfully negotiated, it is used in each subsequent segment until all of the initial SYN options are transmitted.

LO is backward compatible, as is SLO; in both cases, endpoints not supporting the option would not respond with the option, and in both cases the initial SYN is not itself extended.

SLO does modify the three-way handshake because the connection isn't considered completely established until the first data byte is acknowledged. Legacy TCP can establish a connection even in the absence of data. SLO also changes the semantics of the SYN/ACK; for legacy TCP, this completes the active side connection establishment, where in SLO an additional data ACK is required. A connection whose initial SYN options have been confirmed in the SYN/ACK might still fail upon receipt of additional options sent in later SLO segments. This case - of late negotiation fail - is not addressed in the specification.

8.6. LOIC

TCP Long Options by Invalid Checksum is a dual-stack approach that uses two initial SYNS to initiate all updated connections [Yo11]. One SYN negotiates the new option and the other SYN payload contains only the entire options. The negotiation SYN is compliant with existing procedures, but the option SYN has a deliberately incorrect TCP checksum (decremented by 2). A legacy endpoint would discard the segment with the incorrect checksum and respond to the negotiation SYN without the LO option.

Use of the option SYN and its incorrect checksum both interfere with other legacy components. Segments with incorrect checksums will be silently dropped by most middleboxes, including NATs/NAPTs. Use of two SYNs creates side-effects that can delay connections to upgraded endpoints, notably when the option SYN is lost or the SYNs arrive out of order. Finally, by not allowing other options in the negotiation SYN, all connections to legacy endpoints either use no options or require a separate connection attempt (either concurrent or subsequent).

8.7. Problems with Extending the Initial SYN

The key difficulty with most previous proposals is the desire to extend the option space in all TCP segments, including the initial SYN, i.e., SYN with no ACK, typically the first segment of a connection, as well as possibly the SYN/ACK. It has proven difficult to extend space within the segment of the initial SYN in the absence of prior negotiation while maintaining current TCP three-way handshake properties, and it may be similarly challenging to extend the SYN/ACK (depending on asymmetric middlebox assumptions).

A new TCP option cannot extend the Data Offset of a single TCP initial SYN segment, and cannot extend a SYN/ACK in a single segment when considering misbehaving middleboxes. All TCP segments, including the initial SYN and SYN/ACK, may include user data in the payload data [RFC793], and this can be useful for some proposed features such as TCP Fast Open [RFC7413]. Legacy endpoints that ignore the new option would process the payload contents as user data and send an ACK. Once ACK’d, this data cannot be removed from the user stream.

The Reserved TCP header bits cannot be redefined easily, even though three of the six total bits have already been redefined (ECE/CWR [RFC3168] and NS [RFC3540]). Legacy endpoints have been known to reflect received values in these fields; this was safely dealt with for ECN but would be difficult here [RFC3168].

TCP initial SYN (SYN and not ACK) segments can use every other TCP header field except the Acknowledgement number, which is not used because the ACK field is not set. In all other segments, all fields except the three remaining Reserved header bits are actively used. The total amount of available header fields, in either case, is insufficient to be useful in extending the option space.

The representation of TCP options can be optimized to minimize the space needed. In such cases, multiple Kind and Length fields are combined, so that a new Kind would indicate a specific combination of options, whose order is fixed and whose length is indicated by one Length field. Most TCP options use fields whose size is much larger than the required Kind and Length components, so the resulting efficiency is typically insufficient for additional options.

The option space of an initial SYN segment might be extended by using multiple initial segments (e.g., multiple SYNs or a SYN and non-SYN) or based on the context of previous or parallel connections. This method may also be needed to extend space in the SYN/ACK in the presence of misbehaving middleboxes. Because of their potential complexity, these approaches are addressed in separate documents [Bo14][Br14][To18].

Option space cannot be extended in outer layer headers, e.g., IPv4 or IPv6. These layers typically try to avoid extensions altogether, to simplify forwarding processing at routers. Introducing new shim layers to accommodate additional option space would interfere with deep-packet inspection mechanisms that are in widespread use.

As a result, EDO does not attempt to extend the space available for options in TCP initial SYNs. It does extend that space in all other segments (including SYN/ACK), which has always been trivially possible once an option is defined.

9. Implementation Issues

TCP segment processing can involve accessing nonlinear data structures, such as chains of buffers. Such chains are often designed so that the maximum default TCP header (60 bytes) fits in the first buffer. Extending the TCP header across multiple buffers may necessitate buffer traversal functions that span boundaries between buffers. Such traversal can also have a significant performance impact, which is additional rationale for using TCP option space - even extended option space - sparingly.

Although EDO can be large enough to consume the entire segment, it is important to leave space for data so that the TCP connection can make forward progress. It would be wise to limit EDO to consuming no more than MSS-4 bytes of the IP segment, preferably even less (e.g., MSS-128 bytes).

When using the ExID variant for testing and experimentation, either TCP option codepoint (253, 254) is valid in sent or received segments.

Implementers need to be careful about the potential for offload support interfering with this option. The EDO data needs to be passed to the protocol stack as part of the option space, not integrated with the user segment, to allow the offload to independently determine user data segment boundaries and combine them correctly with the extended option data. Some legacy hardware receive offload engines may present challenges in this regard, and may be incompatible with EDO where they incorrectly attempt to process segments with unknown options. Such offload engines are part of the protocol stack and updated accordingly. Issues with incorrect resegmentation by an offload engine can be detected in the same way as middlebox tampering.

10. Security Considerations

It is meaningless to have the Data Offset further exceed the position of the EDO data offset option.

>> When the EDO Extension option is present, the EDO Extension option SHOULD be the last non-null option covered by the TCP Data Offset, because it would be the last option affected by Data Offset.

This also makes it more difficult to use the Data Offset field as a covert channel.

11. IANA Considerations

We request that, upon publication, this option be assigned a TCP Option codepoint by IANA, which the RFC Editor will replace EDO-OPT in this document with codepoint value.

The TCP Experimental ID (ExID) with a 16-bit value of 0x0ED0 (in network standard byte order) has been assigned for use during testing and preliminary experiments.

12. References

12.1. Normative References

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

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

12.2. Informative References

[Al06]   Allman, M., "TCPx2: Don't Fence Me In", draft-allman-tcpx2-hack-00 (work in progress), May 2006.

[Bo14]   Borman, D., "TCP Four-Way Handshake", draft-borman-tcp4way-00 (work in progress), October 2014.

[Br14] Briscoe, B., "Inner Space for TCP Options", draft-briscoe-tcpm-inner-space-01 (work in progress), October 2014.

[Ed08] Eddy, W. and A. Langley, "Extending the Space Available for TCP Options", draft-eddy-tcp-loo-04 (work in progress), July 2008.

[Ho11] Honda, M., Nishida, Y., Raiciu, C., Greenhalgh, A., Handley, M., and H. Tokuda, "Is it still possible to extend TCP", Proc. ACM Sigcomm Internet Measurement Conference (IMC), 2011, pp. 181-194.

[Ko04] Kohler, E., "Extended Option Space for TCP", draft-kohler-tcpm-extopt-00 (work in progress), September 2004.

[Ni15] Nishida, Y., "A-PAWS: Alternative Approach for PAWS", draft-nishida-tcpm-apaws-02 (work in progress), Oct. 2015.

[Ra12] Ramaiah, A., "TCP option space extension", draft-ananth-tcpm-tcpoptext-00 (work in progress), March 2012.

[RFC792] Postel, J., "Internet Control Message Protocol", RFC 792, September 1981.

[RFC1812] Baker, F. (Ed.), "Requirements for IP Version 4 Routers," RFC 1812, June 1995.

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

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

[RFC3234] Carpenter, B. and S. Brim, "Middleboxes: Taxonomy and Issues", RFC 3234, February 2002.

[RFC3540] Spring, N., Wetherall, D., and D. Ely, "Robust Explicit Congestion Notification (ECN) Signaling with Nonces", RFC 3540, June 2003.

[RFC5482] Eggert, L., and F. Gont, "TCP User Timeout Option", RFC 5482, March 2009.

[RFC5925] Touch, J., Mankin, A., and R. Bonica, "The TCP Authentication Option", RFC 5925, June 2010.

[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.

[RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, January 2013.

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

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, December 2014.

[To18] Touch, J., T. Faber, "TCP SYN Extended Option Space Using an Out-of-Band Segment", draft-touch-tcpm-tcp-syn-ext-opt (work in progress), Jan. 2018.

[Yo11] Yourtchenko, A., "Introducing TCP Long Options by Invalid Checksum", draft-yourtchenko-tcp-loic-00 (work in progress), April 2011.

13. Acknowledgments

The authors would like to thank the IETF TCPM WG for their feedback, in particular: Oliver Bonaventure, Bob Briscoe, Ted Faber, John Leslie, Pasi Sarolahti, Richard Scheffenegger, and Alexander Zimmerman.

This work is partly supported by USC/ISI’s Postel Center.

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

Authors’ Addresses

Joe Touch

Manhattan Beach, CA 90266 USA

Phone: +1 (310) 560-0334
Email: touch@strayalpha.com

Wesley M. Eddy
MTI Systems
US

Email: wes@mti-systems.com

Network Working Group N. Khademi
Internet-Draft M. Welzl
Updates: 3168 (if approved) University of Oslo
Intended status: Experimental G. Armitage
Expires: October 5, 2016 Swinburne University of Technology
G. Fairhurst
University of Aberdeen
April 03, 2016

TCP Alternative Backoff with ECN (ABE)
draft-khademi-alternativebackoff-ecn-03

Abstract

This memo provides an experimental update to RFC3168. It updates the TCP sender-side reaction to a congestion notification received via Explicit Congestion Notification (ECN). The updated method reduces cwnd by a smaller amount than TCP does in reaction to loss. The intention is to achieve good throughput when the queue at the bottleneck is smaller than the bandwidth-delay-product of the connection. This is more likely when an Active Queue Management (AQM) mechanism has used ECN to CE-mark a packet, than when a packet was lost. Future versions of this document will discuss SCTP as well as other transports using ECN.

Table of Contents

1. Introduction

Explicit Congestion Notification (ECN) is specified in [RFC3168]. It allows a network device that uses Active Queue Management (AQM) to set the congestion experienced, CE, codepoint in the ECN field of the IP packet header, rather than drop ECN-capable packets when incipient congestion is detected. When an ECN-capable transport is used over a path that supports ECN, it provides the opportunity for flows to improve their performance in the presence of incipient congestion [I-D.AQM-ECN-benefits].

[RFC3168] not only specifies the router use of the ECN field, it also specifies a TCP procedure for using ECN. This states that a TCP sender should treat the ECN indication of congestion in the same way as that of a non-ECN-Capable TCP flow experiencing loss, by halving the congestion window "cwnd" and by reducing the slow start threshold "ssthresh". [RFC5681] stipulates that TCP congestion control sets "ssthresh" to max(FlightSize / 2, 2*SMSS) in response to packet loss. Consequently, a standard TCP flow using this reaction needs significant network queue space: it can only fully utilise a bottleneck when the length of the link queue (or the AQM dropping threshold) is at least the bandwidth-delay product (BDP) of the flow.

A backoff multipler of 0.5 (halving cwnd and sshthresh after packet loss) is not the only available strategy. As defined in [ID.CUBIC], CUBIC multiplies the current cwnd by 0.8 in response to loss (although the Linux implementation of CUBIC has used a multiplier of 0.7 since kernel version 2.6.25 released in 2008). Consequently, CUBIC utilise paths well even when the bottleneck queue is shorter than the bandwidth-delay product of the flow. However, in the case of a DropTail (FIFO) queue without AQM, such less-aggressive backoff increases the risk of creating a standing queue [CODEL2012].

Devices implementing AQM are likely to be the dominant (and possibly only) source of ECN CE-marking for packets from ECN-capable senders. AQM mechanisms typically strive to maintain a small queue length, regardless of the bandwidth-delay product of flows passing through them. Receipt of an ECN CE-mark might therefore reasonably be taken to indicate that a small bottleneck queue exists in the path, and hence the TCP flow would benefit from using a less aggressive backoff multiplier.

Results reported in [ABE2015] show significant benefits (improved throughput) when reacting to ECN-Echo by multiplying cwnd and sstthresh with a value in the range [0.7..0.85]. Section 2 describes the rationale for this change. Section 3 specifies a change to the TCP sender backoff behaviour in response to an indication that CE-marks have been received by the receiver.

2. Discussion

Much of the background to this proposal can be found in [ABE2015]. Using a mix of experiments, theory and simulations with standard NewReno and CUBIC, [ABE2015] recommends enabling ECN and "...letting individual TCP senders use a larger multiplicative decrease factor in reaction to ECN CE-marks from AQM-enabled bottlenecks." Such a change is noted to result in "...significant performance gains in lightly-multiplexed scenarios, without losing the delay-reduction benefits of deploying CoDel or PIE."

2.1. Why use ECN to vary the degree of backoff?

The classic rule-of-thumb dictates a BDP of bottleneck buffering if a TCP connection wishes to optimise path utilisation. A single TCP connection running through such a bottleneck will have opened cwnd up to 2*BDP by the time packet loss occurs. [RFC5681]'s halving of cwnd and ssthresh pushes the TCP connection back to allowing only a BDP of packets in flight -- just enough to maintain 100% utilisation of the network path.

AQM schemes like CoDel [I-D.CoDel] and PIE [I-D.PIE] use congestion notifications to constrain the queuing delays experienced by packets, rather than in response to impending or actual bottleneck buffer exhaustion. With current default delay targets, CoDel and PIE both effectively emulate a shallow buffered bottleneck (section II, [ABE2015]) while allowing short traffic bursts into the queue. This interacts acceptably for TCP connections over low BDP paths, or highly multiplexed scenarios (lmany concurrent TCP connections). However, it interacts badly with lightly-multiplexed cases (few concurrent connections) over high BDP paths. Conventional TCP backoff in such cases leads to gaps in packet transmission and under-utilisation of the path.

In an ideal world, the TCP sender would adapt its backoff strategy to match the effective depth at which a bottleneck begins indicating congestion. In the practical world, [ABE2015] proposes using the existence of ECN CE-marks to infer whether a path’s bottleneck is AQM-enabled (shallow queue) or classic DropTail (deep queue), and adjust backoff accordingly. This results in a change to [RFC3168], which recommended that TCP senders respond in the same way following indication of a received ECN CE-mark and a packet loss, making these equivalent signals of congestion. (The idea to change this behaviour pre-dates ABE. [ICC2002] also proposed using ECN CE-marks to modify TCP congestion control behaviour, using a larger multiplicative decrease factor in conjunction with a smaller additive increase factor to deal with RED-based bottlenecks that were not necessarily configured to emulate a shallow queue.)

[RFC7567] states that "deployed AQM algorithms SHOULD support Explicit Congestion Notification (ECN) as well as loss to signal congestion to endpoints" and [I-D.AQM-ECN-benefits] encourages this deployment. Apple recently announced their intention to enable ECN in iOS 9 and OS X 10.11 devices [WWDC2015]. By 2014, server-side ECN negotiation was observed to be provided by the majority of the top million web servers [PAM2015], and only 0.5% of websites incurred additional connection setup latency using RFC3168-compliant ECN-fallback mechanisms.

2.2. Choice of ABE multiplier

ABE decouples a TCP sender’s reaction to loss and ECN CE-marks. The description respectively uses beta_{loss} and beta_{ecn} to refer to the multiplicative decrease factors applied in response to packet loss and in response to an indication of a received CN CE-mark on an ECN-enabled TCP connection (based on the terms used in [ABE2015]).

For non-ECN-enabled TCP connections, no ECN CE-marks are received and only beta_{loss} applies.

In other words, in response to detected loss:

FlightSize_(n+1) = FlightSize_n * beta_{loss}

and in response to an indication of a received ECN CE-mark:

FlightSize_(n+1) = FlightSize_n * beta_{ecn}

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

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

The Internet has already been running with at least two different beta_{loss} values for several years: the value in [RFC5681] is 0.5, and Linux CUBIC uses 0.7. ABE proposes no change to beta_{loss} used by any current TCP implementations.

beta_{ecn} depends on how we want to optimise the reponse of a TCP connection to shallow AQM marking thresholds. beta_{loss} reflects the preferred response of each TCP algorithm when faced with exhaustion of buffers (of unknown depth) signalled by packet loss. Consequently, for any given TCP algorithm the choice of beta_{ecn} is likely to be algorithm-specific, rather than a constant multiple of the algorithm's existing beta_{loss}.

A range of experiments (section IV, [ABE2015]) with NewReno and CUBIC over CoDel and PIE in lightly multiplexed scenarios have explored this choice of parameter. These experiments indicate that CUBIC connections benefit from beta_{ecn} of 0.85 (cf. beta_{loss} = 0.7), and NewReno connections see improvements with beta_{ecn} in the range 0.7 to 0.85 (c.f., beta_{loss} = 0.5).

3. NEW: Updating the Sender-side ECN Reaction

This section specifies an experimental update to [RFC3168].

3.1. RFC 2119

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].

3.2. Update to RFC 3168

This document specifies an update to the TCP sender reaction that follows when the TCP receiver signals that ECN CE-marked packets have been received.

The first paragraph of Section 6.1.2, "The TCP Sender", in [RFC3168] contains the following text:

"If the sender receives an ECN-Echo (ECE) ACK packet (that is, an ACK packet with the ECN-Echo flag set in the TCP header), then the sender knows that congestion was encountered in the network on the path from the sender to the receiver. The indication of congestion should be treated just as a congestion loss in non-ECN-Capable TCP. That is, the TCP source halves the congestion window "cwnd" and reduces the slow start threshold "ssthresh"."

This memo updates this by replacing it with the following text:

"If the sender receives an ECN-Echo (ECE) ACK packet (that is, an ACK packet with the ECN-Echo flag set in the TCP header), then the sender knows that congestion was encountered in the network on the path from the sender to the receiver. This indication of congestion could be treated in the same way as a congestion loss, however reception of the ECN-Echo flag SHOULD produce a reduction in FlightSize that is less than the reduction had the flow experienced loss. The reduction needs to be sufficient to allow flows sharing a bottleneck to increase their share of the capacity. This reduction MUST be less than 0.85 (at least a 15% reduction).

An ECN-capable network device cannot eliminate the possibility of loss, because a drop may occur due to a traffic burst exceeding the instantaneous available capacity of a network buffer or as a result of the AQM algorithm (overload protection mechanisms, etc [RFC7567]). Whatever the cause of loss, detection of a missing packet needs to trigger the standard loss-based congestion control response. This explicitly does not update this behaviour.

In addition, this document RECOMMENDS that experimental deployments method multiply the FlightSize by 0.8 and reduce the slow start threshold ’ssthresh’ in response to reception of a TCP segment that sets the ECN-Echo flag."

3.3. Status of the Update

This update is a sender-side only change. Like other changes to congestion-control algorithms it does not require any change to the TCP receiver or to network devices (except to enable an ECN-marking algorithm [RFC3168] [RFC7567]). If the method is only deployed by some TCP senders, and not by others, the senders that use this method can gain advantage, possibly at the expense of other flows that do not use this updated method. This advantage applies only to ECN-marked packets and not to loss indications. Hence, the new method can not lead to congestion collapse.

The present specification has been assigned an Experimental status, to provide Internet deployment experience before being proposed as a Standards-Track update.

4. Acknowledgements

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

The authors would like to thank the following people for their contributions to [ABE2015]: Chamil Kulatunga, David Ros, Stein Gjessing, Sebastian Zander. Thanks to (in alphabetical order) Bob Briscoe, John Leslie, Dave Taht and the TCPM WG for providing valuable feedback on this document.

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

5. IANA Considerations

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

This memo includes no request to IANA.

6. Security Considerations

The described method is a sender-side only transport change, and does not change the protocol messages exchanged. The security considerations of RFC 3819 therefore still apply.

This document describes a change to TCP congestion control with ECN that will typically lead to a change in the capacity achieved when flows share a network bottleneck. Similar unfairness in the way that capacity is shared is also exhibited by other congestion control mechanisms that have been in use in the Internet for many years (e.g., CUBIC [ID.CUBIC]). Unfairness may also be a result of other factors, including the round trip time experienced by a flow. This advantage applies only to ECN-marked packets and not to loss indications, and will therefore not lead to congestion collapse.

7. References

7.1. Normative References

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

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <http://www.rfc-editor.org/info/rfc3168>.

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

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

7.2. Informative References

[ABE2015] Khademi, N., Welzl, M., Armitage, G., Kulatunga, C., Ros, D., Fairhurst, G., Gjessing, S., and S. Zander, "Alternative Backoff: Achieving Low Latency and High Throughput with ECN and AQM", CAIA Technical Report CAIA-TR-150710A, Swinburne University of Technology, July 2015, <http://caia.swin.edu.au/reports/150710A/CAIA-TR-150710A.pdf>.

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

[I-D.AQM-ECN-benefits]
Fairhurst, G. and M. Welzl, "The Benefits of using Explicit Congestion Notification (ECN)", Internet-draft, IETF work-in-progress draft-ietf-aqm-ecn-benefits-08, November 2015.

[I-D.CoDel]
Nichols, K., Jacobson, V., McGregor, V., and J. Iyengar, "The Benefits of using Explicit Congestion Notification (ECN)", Internet-draft, IETF work-in-progress draft-ietf-aqm-codel-02, December 2015.

[I-D.PIE] Pan, R., Natarajan, P., Baker, F., White, G., VerSteeg, B., Prabhu, M., Piglione, C., and V. Subramanian, "PIE: A Lightweight Control Scheme To Address the Bufferbloat Problem", Internet-draft, IETF work-in-progress draft-ietf-aqm-pie-03, November 2015.

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

[ID.CUBIC]
Rhee, I., Xu, L., Ha, S., Zimmermann, A., Eggert, L., and R. Scheffenegger, "CUBIC for Fast Long-Distance Networks", Internet-draft, IETF work-in-progress draft-ietf-tcpm-cubic-00, June 2015.

[PAM2015] Trammell, B., Kuhlewind, M., Boppart, D., Learmonth, I., Fairhurst, G., and R. Scheffenegger, "Enabling Internet-wide Deployment of Explicit Congestion Notification", Proceedings of the 2015 Passive and Active Measurement Conference, New York, March 2015, <http://ecn.ethz.ch/ecn-pam15.pdf>.

[WWDC2015]
Lakhera, P. and S. Cheshire, "Your App and Next Generation Networks", Apple Worldwide Developers Conference 2015, San Francisco, USA, June 2015, <https://developer.apple.com/videos/wwdc/2015/?id=719>.

Authors' Addresses

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

Email: naeemk@ifi.uio.no

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

Email: michawe@ifi.uio.no

Grenville Armitage
Centre for Advanced Internet Architectures
Swinburne University of Technology
PO Box 218
John Street, Hawthorn
Victoria 3122
Australia

Email: garmitage@swin.edu.au

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

Email: gorry@erg.abdn.ac.uk

Network Working Group H. Kitamura
Internet-Draft NEC Corporation
Intended status: Standards Track S. Ata
Expires: April 2016 Osaka City University
M. Murata
Osaka University
October 16, 2015

"Sharp Close": Elimination of TIME-WAIT state of TCP connections
<draft-kitamura-tcp-sharp-close-02.txt>

Abstract

This document describes an idea "Sharp Close" that eliminates or minimizes TIME-WAIT state of TCP connections.

In the current TCP specification ([RFC0793]), there are some inappropriate or not up-to-date functions. Here we focus and discuss on TCP TIME-WAIT state function.

TIME-WAIT is the last state of TCP connections of Active Close side nodes. After TCP connections are effectively closed, state of them move to TIME-WAIT state. After TIME-WAIT state is finished, resources of connections are released. This means that even if connections are effectively finished, resources of connections are NOT released. The TIME-WAIT state prevents from releasing them.

From the viewpoints of current high-speed and high-multiplicity communication styles, it is thought that TIME-WAIT state is one of evil functions.

In order to provide efficient communications that match current styles, an idea "Sharp Close" that eliminates or minimizes TIME-WAIT state of TCP connections is proposed.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

This document describes an idea "Sharp Close" that eliminates or minimizes TIME-WAIT state of TCP connections.

In the current TCP specification ([RFC0793]), there are some inappropriate or not up-to-date functions. Here we focus and discuss on TCP TIME-WAIT state function.

TIME-WAIT is the last state of TCP connections of Active Close side nodes. After TCP connections are effectively closed, state of them move to TIME-WAIT state. [RFC0793] defines that the connections stay there 2MSL(Maximum Segment Lifetime) seconds. (2MSL = 240 sec.)

After TIME-WAIT state is finished, resources of connections are released. This means that even if connections are effectively finished, resources of connections are NOT released. The TIME-WAIT state prevents from releasing them.

From the viewpoints of current high-speed and high-multiplicity communication styles that require highly resource recycling, it is thought that TIME-WAIT state is one of evil functions.

In order to provide efficient communications that match current styles, an idea "Sharp Close" that eliminates or minimizes TIME-WAIT state of TCP connections is proposed.

In the following sections, analysis of current TIME-WAIT state and design of "Sharp Close" etc. are described.

2. Analysis of current TIME-WAIT state

```
                ACTIVE CLOSE   PASSIVE CLOSE
                     side         side
                       |            |
            ESTABLISH  |            |  ESTABLISH
      _________________|_ FIN       |
                       | \__        |
                       |    \__     |
                       |       \__  |
           FIN-WAIT-1  |          \_|__________________
                       |        __/ |
                       |     __/ACK |  CLOSE-WAIT
                       |  __/      _|__________________
      _________________|_/      __/ |
                       |     __/FIN |
           FIN-WAIT-2  |  __/       |
      _________________|_/          |  LAST-ACK
        |              | \_ACK      |
        |              |    \__     |
        |              |       \__  |
      2MSL  TIME-WAIT  |          \_|__________________
        |              |            |
        |              |            |
        |              |            |  CLOSED
      __V______________|            |
                       |            |
               CLOSED  |            |
                       |            |

              Fig. 1  Current ACTIVE-PASSIVE Close Sequence
```

```
                ACTIVE CLOSE   ACTIVE CLOSE
                     side         side
                       |            |
            ESTABLISH  |            |  ESTABLISH
      _________________|_ FIN       |
                       | \__    FIN_|__________________
                       |    \__ __/ |
           FIN-WAIT-1  |     __X__  |  FIN-WAIT-1
                       |  __/     \_|__________________
      _________________|_/      __/ |
                       | \_ACK_/ACK |
              CLOSING  |  __\/_     |  CLOSING
      _________________|_/     \__  |
        |              |          \_|__________________
        |              |            |               |
        |              |            |               |
      2MSL  TIME-WAIT  |            |               |
        |              |            |  TIME-WAIT  2MSL
        |              |            |               |
        |              |            |               |
      __V______________|            |               |
                       |            |_______________V__
               CLOSED  |            |
                       |            |  CLOSED
                       |            |
```

Fig. 2 Current ACTIVE-ACTIVE Close Sequence

Fig. 1 and Fig. 2 show Close Sequence that is defined by current specification [RFC0793]. TCP connections on ACTIVE CLOSE node (that initiates sending FIN) side reach TIME-WAIT as a last state. They stay there 2MSL seconds.

Table 1 Actual 2MSL values used by major OS implementation.

RFC/OS	2MSL value
[RFC0793]	240 sec.
Windows2000	240 sec.
Windows (after Win2K)	120 sec.
Unix/Linux	60 sec.

Table 1 shows actual 2MSL values that are surveyed by authors.

[RFC0793] says "For this specification the MSL is taken to be 2 minutes."

Since 240 sec. ([RFC0793]) is long time, recent major OSes adopt rather shorter time.

However, from the viewpoints of current communication styles that require highly resource recycling, TIME-WAIT time is still too long.

Now, it is almost thought that staying at TIME-WAIT state is waste of time.

3. Why TIME-WAIT state is needed?

Basically, TIME-WAIT state is designed for !fail-safe! purpose.

If it is assumed that packets transferring order is not changed, all of !data! packets from a corresponding node are received when FIN-WAIT-2 state is finished (responding FIN packet is received) and no !data! packets will not be received after that.

At TIME-WAIT state, an ACTIVE CLOSE node waits for a ’resending’ !control! packet FIN only from the corresponding node for the case of the sent ACK (for the FIN) is lost. (No !data! packets are waited for.)

Only when the last sent ACK from the ACTIVE CLOSE node is lost, ’resending’ control packet FIN from the corresponding node is issued.

It is rare case to happen this event at current stable network environment.

Since all data from the corresponding node is received by the ACTIVE CLOSE node, it is less significant issue to wait for ’resending’ FIN packet.

If ’resending’ FIN is NOT waited at ACTIVE CLOSE node and ’resending’ FIN is issued from the corresponding node, significant problem will NOT be happened, only RST packet (to notify receiving unexpected packet) will be issued from the ACTIVE CLOSE node.

4. Design of "Sharp Close" (elimination of TIME-WAIT state)

```
                ACTIVE CLOSE   PASSIVE CLOSE
                     side         side
                       |            |
            ESTABLISH  |            |  ESTABLISH
      _________________|_ FIN       |
                       | \__        |
                       |    \__     |
                       |       \__  |
           FIN-WAIT-1  |          \_|__________________
                       |        __/ |
                       |     __/ACK |  CLOSE-WAIT
                       |  __/      _|__________________
      _________________|_/      __/ |
                       |     __/FIN |
           FIN-WAIT-2  |  __/       |
      _________________|_/          |  LAST-ACK
       (NO) TIME-WAIT  | \_ACK      |
                       |    \__     |
               CLOSED  |       \__  |
                       |          \_|__________________
                       |            |
                       |            |  CLOSED
                       |            |
```

         Fig. 3 (Proposed) Sharp ACTIVE-PASSIVE Close Sequence

```
                ACTIVE CLOSE   ACTIVE CLOSE
                     side         side
                       |            |
            ESTABLISH  |            |  ESTABLISH
      _________________|_ FIN       |
                       | \__    FIN_|__________________
                       |    \__ __/ |
           FIN-WAIT-1  |     __X__  |  FIN-WAIT-1
                       |  __/     \_|__________________
      _________________|_/      __/ |
                       | \_ACK_/ACK |
              CLOSING  |  __\/_     |  CLOSING
      _________________|_/     \__  |
       (NO) TIME-WAIT  |          \_|__________________
                       |            | (NO) TIME-WAIT
               CLOSED  |            |
                       |            |  CLOSED
```

         Fig. 4 (Proposed) Sharp ACTIVE-ACTIVE Close Sequence

It is easy to design "Sharp Close" function. "Sharp Close" function is achieved by eliminating or minimizing TIME-WAIT state of TCP connections.

Fig. 3 and Fig. 4. show Close Sequence that is defined by "Sharp Close" function.

5. Eliminate TIME-WAIT state by setsockopt()

Under current implementation, TIME-WAIT (close()) action can be controlled by setsockopt() function.

SO_LINGER option of setsockopt() can eliminate TIME-WAIT state and close connections immediately.

Concrete procedures how to eliminate TIME-WAIT:

Fig. 5 shows struct socket in <sys/socket.h>

```
struct linger {
       int l_onoff;    /* linger active */
       int l_linger;   /* how many seconds to linger for */
};
```

Fig. 5. struct linger

By using the following shown procedures, TIME-WAIT state is eliminated and connections are closed immediately.

1: makes linger active(on)
```
l_onoff = on;
```

2: sets linger time to 0
```
l_linger = 0 ;
```

It is possible to eliminate TIME-WAIT state by these procedures. However, this behavior is "NOT default" operation. In order to utilize this feature, it is necessary to modify huge number of communication applications.

Furthermore, this feature is not implemented on every existing OSes and it is not always possible to eliminate TIME-WAIT state on every OSes.

6. Security Considerations

Goals of the proposed idea ("Sharp Close") are to eliminate or minimize TIME-WAIT state by default on OS kernel level. From functional viewpoints, the same concept to eliminate TIME-WAIT state is already implemented by using LINGER option of setsockopt() function. It is not default operation, however it has already implemented and worked.

So, there are no new Security Consideration issues that should be discussed here.

7. IANA Considerations

This document does not require any resource assignments to IANA.

Acknowledgment

A part of this work is supported by the program: SCOPE (Strategic Information and Communications R&D Promotion Programme) operated by Ministry of Internal Affairs and Communications of JAPAN.

Appendix A. Implementations

Currently, above described "Sharp Close" functions have been implemented and verified under the following OS.

Ubuntu 13.04 (kernel 3.8.13.8)

References

Normative References

[RFC0793] "Transmission Control Protocol", RFC 0793, September 1981

[RFC3513] R. Hinden and S.Deering, "Internet Protocol Version 6 (IPv6) Addressing Architecture", RFC 3513, April 2003

[RFC3493] R. Gilligan, S. Thomson, J. Bound, J. McCann and W. Stevens, "Basic Socket Interface Extensions for IPv6", RFC3493, February 2003

Informative References

[RFC3542] W. Stevens, M. Thomas, E. Nordmark and T. Jinmei, "Advanced Sockets Application Program Interface (API) for IPv6", RFC 3542, May 2003

Authors' Addresses

Hiroshi Kitamura
Cyber Security Strategy Division / Cloud System Research Laboratories,
NEC Corporation
7-1, Shiba 5-chome, Minato-ku, Tokyo 108-8001, JAPAN
Phone: +81 3 3798 0563
Email: kitamura@da.jp.nec.com

Shingo Ata
Graduate School of Engineering, Osaka City University
3-3-138, Sugimoto, Sumiyoshi-Ku, Osaka 558-8585, JAPAN
Phone: +81 6 6605 2191
Fax: +81 6 6605 2191
Email: ata@info.eng.osaka-cu.ac.jp

Masayuki Murata
Graduate School of Information Science and Technology, Osaka Univ.
1-5 Yamadaoka, Suita, Osaka 565-0871, JAPAN
Phone: +81 6 6879 4542
Fax: +81 6 6879 4544
Email: murata@ist.osaka-u.ac.jp

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

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

October 19, 2015

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

Abstract

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

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Explicit Congestion Notification (ECN) [RFC3168] is a mechanism where network nodes can mark IP packets instead of dropping them to indicate incipient congestion to the end-points. Receivers with an ECN-capable transport protocol feed back this information to the sender. ECN is specified for TCP in such a way that only one feedback signal can be transmitted per Round-Trip Time (RTT). Recently, proposed mechanisms like Congestion Exposure (ConEx [I-D.ietf-conex-abstract-mech]) or DCTCP [I-D.bensley-tcpm-dctcp] need more accurate ECN feedback information whenever more than one marking is received in one RTT. A fuller treatment of the motivation for this specification is given in the associated requirements document [RFC7560].

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

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

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

1.1. Document Roadmap

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

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

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

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

1.2. Goals

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

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

1.3. Experiment Goals

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

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

1.4. Terminology

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

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

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

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

Pure ACK: A TCP acknowledgement without a data payload.

TCP client: The TCP stack that originates a connection.

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

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

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

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

1.5. Recap of Existing ECN feedback in IP/TCP

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

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

Table 1: The ECN Field in the IP Header

In the TCP header the first two bits in byte 14 are defined as flags for the use of ECN (CWR and ECE in Figure 1 [RFC3168]). A TCP client indicates it supports ECN by setting ECE=CWR=1 in the SYN, and an ECN-enabled server confirms ECN support by setting ECE=1 and CWR=0 in the SYN/ACK. On reception of a CE-marked packet at the IP layer, the Data Receiver starts to set the Echo Congestion Experienced (ECE) flag continuously in the TCP header of ACKs, which ensures the signal is received reliably even if ACKs are lost. The TCP sender confirms that it has received at least one ECE signal by responding with the congestion window reduced (CWR) flag, which allows the TCP receiver to stop repeating the ECN-Echo flag. This always leads to a full RTT of ACKs with ECE set. Thus any additional CE markings arriving within this RTT cannot be fed back.

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

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

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

2. AccECN Protocol Overview and Rationale

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

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

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

- 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;

- a supplementary part using a new AccECN TCP Option that provides additional feedback on the number of bytes that arrive marked with each of the three ECN codepoints (not just CE marks). This provides greater resilience against ACK loss than the essential feedback, but it is more likely to suffer from middlebox interference.

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

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

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

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

2.1. Capability Negotiation

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

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

2.2. Feedback Mechanism

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

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

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

2.3. Delayed ACKs and Resilience Against ACK Loss

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

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

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

If the AccECN Option is not available, e.g. it is being stripped by a middlebox, the AccECN protocol will only feed back information on CE markings (using the ACE field). Although not ideal, this will be sufficient, because it is envisaged that neither ECT(0) nor ECT(1) will ever indicate more severe congestion than CE, even though future uses for ECT(0) or ECT(1) are still unclear. 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, even though it is not allowed to be ECN-capable according to RFC 3168. However, middleboxes have been known to overwrite the ECN IP field as if it is still part of the old Type of Service (ToS) field. If a TCP client has set the SYN to Not-ECT, but receives CE feedback, it can detect such middlebox interference and send Not-ECT for the rest of the connection (see [I-D.kuehlewind-tcpm-ecn-fallback] for the detailed fall-back behaviour).

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

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

3. AccECN Protocol Specification

3.1. Negotiation during the TCP handshake

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

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

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

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

If after the normal TCP timeout the TCP client has not received a SYN/ACK to acknowledge its SYN, the SYN might just have been lost, e.g. due to congestion, or a middlebox might be blocking segments with the AccECN flags. To expedite connection setup, the host SHOULD fall back to NS=CWR=ECE=0 on the retransmission of the SYN. It would make sense to also remove any other experimental fields or options on the SYN in case a middlebox might be blocking them, although the required behaviour will depend on the specification of the other option(s) and any attempt to co-ordinate fall-back between different modules of the stack. Implementers MAY use other fall-back strategies if they are found to be more effective (e.g. attempting to retransmit a second AccECN segment before fall-back, falling back to classic ECN feedback rather than non-ECN, and/or caching the result of a previous attempt to access the same host while negotiating AccECN).

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

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

Ac: More *Ac*curate ECN Feedback

N: ECN-*N*once [RFC3540]

E: *E*CN [RFC3168]

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

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

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

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

1. The top block shows the case already described where both endpoints support AccECN and how the TCP server (B) indicates congestion feedback.

2. The second block shows the cases where the TCP client (A) supports AccECN but the TCP server (B) supports some earlier variant of TCP feedback, indicated in its SYN/ACK. Therefore, as soon as an AccECN-capable TCP client (A) receives the SYN/ACK shown it MUST set both its half connections into the feedback mode shown in the rightmost column.

3. The third block shows the cases where the TCP server (B) supports AccECN but the TCP client (A) supports some earlier variant of TCP feedback, indicated in its SYN. Therefore, as soon as an AccECN-enabled TCP server (B) receives the SYN shown, it MUST set both its half connections into the feedback mode shown in the rightmost column.

4. The fourth block displays combinations that are not valid or currently unused and therefore both ends MUST fall-back to Not ECN for both half connections. Especially the first case (marked 'broken') where all bits set in the SYN are reflected by the receiver in the SYN/ACK, which happens quite often if the TCP

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

The following exceptional cases need some explanation:

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

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

3.2. AccECN Feedback

Each Data Receiver maintains four counters, r.cep, r.ceb, r.e0b and r.e1b. The CE packet counter (r.cep), counts the number of packets the host receives with the CE code point in the IP ECN field, including CE marks on control packets without data. r.ceb, r.e0b and r.e1b count the number of TCP payload bytes in packets marked respectively with the CE, ECT(0) and ECT(1) codepoint in their IP-ECN field. When a host first enters AccECN mode, it initialises its counters to r.cep = 6, r.e0b = 1 and r.ceb = r.e1b.= 0 (see Appendix A.5). Non-zero initial values are used to be distinct from cases where the fields are incorrectly zeroed (e.g. by middleboxes).

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

3.2.1. The ACE Field

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

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

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

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

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

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

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

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

3.2.2. Safety against Ambiguity of the ACE Field

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

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

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

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

3.2.3. The AccECN Option

The AccECN Option is defined as shown below in Figure 3. It consists of three 24-bit fields that provide the 24 least significant bits of the r.e0b, r.ceb and r.e1b counters, respectively. The initial 'E' of each field name stands for 'Echo'.

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

Figure 3: The AccECN Option

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

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

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

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

Length=11: EE0B, ECEB, EE1B

Length=8: EE0B, ECEB

Length=5: EE0B

Length=2: (empty)

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

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

3.2.4. Path Traversal of the AccECN Option

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

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

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

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

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

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

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

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

3.2.5. Usage of the AccECN TCP Option

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

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

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

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

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

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

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

Implementation note: sending an AccECN Option each time a different counter changes and including a full-length AccECN Option on every delayed ACK will satisfy the requirements described above and might be the easiest implementation, as long as sufficient space is available in each ACK (in total and in the option space).

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

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

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

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

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

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

4. Interaction with Other TCP Variants

This section is informative, not normative.

4.1. Compatibility with SYN Cookies

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

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

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

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

4.2. Compatibility with Other TCP Options and Experiments

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

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

4.3. Compatibility with Feedback Integrity Mechanisms

The ECN Nonce [RFC3540] is an experimental IETF specification intended to allow a sender to test whether ECN CE markings (or losses) introduced in one network are being suppressed by the receiver or anywhere else in the feedback loop, such as another network or a middlebox. The ECN nonce has not been deployed as far as can be ascertained. The nonce would now be nearly impossible to deploy retrospectively, because to catch a misbehaving receiver it relies on the receiver volunteering feedback information to incriminate itself. A receiver that has been modified to misbehave can simply claim that it does not support nonce feedback, which will seem unremarkable given so many other hosts do not support it either.

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

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

- The Data Sender can test the integrity of the receiver's ECN (or loss) feedback by occasionally setting the IP-ECN field to a value normally only set by the network (and/or deliberately leaving a sequence number gap). Then it can test whether the Data Receiver's feedback faithfully reports what it expects [I-D.moncaster-tcpm-rcv-cheat]. Unlike the ECN Nonce, this approach does not waste the ECT(1) codepoint in the IP header, it does not require standardisation and it does not rely on misbehaving receivers volunteering to reveal feedback information that allows them to be detected. However, setting the CE mark by the sender might conceal actual congestion feedback from the network and should therefore only be done sparsely.

- Networks generate congestion signals when they are becoming congested, so they are more likely than Data Senders to be concerned about the integrity of the receiver's feedback of these signals. A network can enforce a congestion response to its ECN markings (or packet losses) using congestion exposure (ConEx) audit [I-D.ietf-conex-abstract-mech]. Whether the receiver or a downstream network is suppressing congestion feedback or the sender is unresponsive to the feedback, or both, ConEx audit can neutralise any advantage that any of these three parties would otherwise gain.

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

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

5. Protocol Properties

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

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

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

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

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

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

Resilience: All information is provided based on counters. Therefore if ACKs are lost, the counters on the first ACK following the losses allows the Data Sender to immediately recover the number of the ECN markings that it missed.

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

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

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

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

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

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

Backward Compatibility: If the AccECN Option is stripped by a middlebox, AccECN still provides basic congestion feedback in the ACE field. Further, AccECN can be used to detect mangling of the IP ECN field; mangling of the TCP ECN flags; blocking of ECT-marked segments; and blocking of segments carrying the AccECN Option. It can detect these conditions during TCP’s 3WHS so that it can fall back to operation without ECN and/or operation without the AccECN Option.

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

6. IANA Considerations

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

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

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

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

7. Security Considerations

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

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

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

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

8. Acknowledgements

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

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

9. Comments Solicited

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

10. References

10.1. Normative References

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

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <http://www.rfc-editor.org/info/rfc3168>.

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

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

10.2. Informative References

[I-D.bensley-tcpm-dctcp]
Bensley, S., Eggert, L., Thaler, D., Balasubramanian, P., and G. Judd, "Microsoft's Datacenter TCP (DCTCP): TCP Congestion Control for Datacenters", draft-bensley-tcpm-dctcp-05 (work in progress), July 2015.

[I-D.ietf-conex-abstract-mech]
Mathis, M. and B. Briscoe, "Congestion Exposure (ConEx) Concepts, Abstract Mechanism and Requirements", draft-ietf-conex-abstract-mech-13 (work in progress), October 2014.

[I-D.kuehlewind-tcpm-ecn-fallback]
Kuehlewind, M. and B. Trammell, "A Mechanism for ECN Path Probing and Fallback", draft-kuehlewind-tcpm-ecn-fallback-01 (work in progress), September 2013.

[I-D.moncaster-tcpm-rcv-cheat]
Moncaster, T., Briscoe, B., and A. Jacquet, "A TCP Test to Allow Senders to Identify Receiver Non-Compliance", draft-moncaster-tcpm-rcv-cheat-03 (work in progress), July 2014.

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

[RFC4987] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", RFC 4987, DOI 10.17487/RFC4987, August 2007, <http://www.rfc-editor.org/info/rfc4987>.

[RFC5562] Kuzmanovic, A., Mondal, A., Floyd, S., and K. Ramakrishnan, "Adding Explicit Congestion Notification (ECN) Capability to TCP's SYN/ACK Packets", RFC 5562, DOI 10.17487/RFC5562, June 2009, <http://www.rfc-editor.org/info/rfc5562>.

[RFC5925] Touch, J., Mankin, A., and R. Bonica, "The TCP Authentication Option", RFC 5925, DOI 10.17487/RFC5925, June 2010, <http://www.rfc-editor.org/info/rfc5925>.

[RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <http://www.rfc-editor.org/info/rfc6824>.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <http://www.rfc-editor.org/info/rfc7413>.

[RFC7560] Kuehlewind, M., Ed., Scheffenegger, R., and B. Briscoe, "Problem Statement and Requirements for Increased Accuracy in Explicit Congestion Notification (ECN) Feedback", RFC 7560, DOI 10.17487/RFC7560, August 2015, <http://www.rfc-editor.org/info/rfc7560>.

Appendix A. Example Algorithms

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

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

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

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

```
DIVOPT = 2^24
```

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

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

where '%' is the modulo operator.

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

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

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

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

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

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

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

A.2.1. Safety Algorithm without the AccECN Option

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

```
DIVACE = 2^3
```

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

```
ACE = r.cep % DIVACE.
```

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

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

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

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

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

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

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

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

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

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

A.2.2. Safety Algorithm with the AccECN Option

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

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

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

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

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

- o if d.cep=0, dSafer.cep=8 and d.ceb=1,460, then s=infinity and sSafer=182.5.
  Therefore even though the average size of 8 data segments is unlikely to have been as small as MSS/8, d.cep cannot have been correct, because it would imply an average segment size greater than the MSS.

- o if d.cep=2, dSafer.cep=10 and d.ceb=1,460, then s=730 and sSafer=146.
  Therefore d.cep is safe enough, because the average size of 10 data segments is unlikely to have been as small as MSS/10.

- o if d.cep=7, dSafer.cep=15 and d.ceb=10,200, then s=1,457 and sSafer=680.
  Therefore d.cep is safe enough, because the average data segment size is more likely to have been just less than one MSS, rather than below MSS/2.

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

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

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

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

```
s_ave ~= flightsize / packets_in_flight,
```

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

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

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

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

A.4. Example Algorithm to Beacon AccECN Options

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

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

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

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

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

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

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

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

A Data Sender in AccECN mode can infer the amount of TCP payload data arriving at the receiver marked Not-ECT from the difference between the amount of newly ACKed data and the sum of the bytes with the other three markings, d.ceb, d.e0b and d.e1b. Note that, because r.e0b is initialised to 1 and the other two counters are initialised to 0, the initial sum will be 1, which matches the initial offset of the TCP sequence number on completion of the 3WHS.

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

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

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

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

Feedback all four ECN codepoints on the SYN/ACK: The last two negotiation combinations in Table 2 could also be used to indicate AccECN support and to feedback that the arriving SYN was ECT(0) or ECT(1). This could be used to probe the client to server path for incorrect forwarding of the ECN field [I-D.kuehlewind-tcpm-ecn-fallback]. Note, however, that it would be unremarkable if ECN on the SYN was zeroed by security devices, given RFC 3168 prohibited ECT on SYN because it enables DoS attacks.

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

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

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

1. Currently it is specified that the receiver ‘SHOULD’ use Change-Triggered ACKs. It is controversial whether this ought to be a ‘MUST’ instead. A ‘SHOULD’ would leave the Data Sender uncertain whether it can rely on the timing and ordering information in ACKs. If the sender guesses wrongly, it will probably introduce at least 1RTT of delay before it can use this timing information. Ironically it will most likely be wanting this information to reduce ramp-up delay. A ‘MUST’ could make it hard to implement AccECN in offload hardware. However, it is not known whether AccECN would be hard to implement in such hardware even with a ‘SHOULD’ here. For instance, was it hard to offload DCTCP to hardware because of change-triggered ACKs, or was this just one of many reasons? The choice between MUST and SHOULD here is critical. Before that choice is made, a clear use-case for certainty of timing and ordering information is needed, plus well-informed discussion about hardware offload constraints.

2. There is possibly a concern that a receiver could deliberately omit the AccECN Option pretending that it had been stripped by a middlebox. No known way can yet be contrived to take advantage of this downgrade attack, but it is mentioned here in case someone else can contrive one.

3. The s.cep counter might increase even if the s.ceb counter does not (e.g. due to a CE-marked control packet). The sender’s response to such a situation is considered out of scope, because this ought to be dealt with in whatever future specification allows ECN-capable control packets. However, it is possible that the situation might arise even if the sender has not sent ECN-

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

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

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

From 04 to 05::

* Corrected ambiguity between Classic ECN and Classic ECN feedback throughout

* Changed MUST to SHOULD send AccECN option on SYN/ACK last ACK of 3WHS and first data segment from client, to allow for cached knowledge of option traversal problems.

* Removed duplication of normative language about sending a full-length option in the sections on "The AccECN Option" and "Usage of the AccECN Option", and mutually cross referenced.

* Acknowledged Koen De Schepper and Praveen Balasubramanian

* Noted in Appendix that algo to beacon a full-length option is work-in-progress

* Editorial corrections and clarifications throughout

Authors' Addresses

Bob Briscoe
Simula Research Laboratory

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

Mirja Kuehlewind
ETH Zurich
Gloriastrasse 35
Zurich 8092
Switzerland

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

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

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

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: April 19, 2016

Y. Nishida
GE Global Research

October 17, 2015

A-PAWS: Alternative Approach for PAWS
draft-nishida-tcpm-apaws-02

Abstract

This documents describe a technique called A-PAWS which can provide protection against old duplicates segments like PAWS. While PAWS requires TCP to set timestamp options in all segments in a TCP connection, A-PAWS supports the same feature without using timestamps. A-PAWS is designed to be used complementary with PAWS. TCP needs to use PAWS when it is necessary and activates A-PAWS only when it is safe to use. Without impairing the reliability and the robustness of TCP, A-PAWS can provide more option space to other TCP extensions.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

PAWS (Protect Against Wrapped Sequences) defined in [RFC1323] is a technique that can identify old duplicate segments in a TCP connection. An old duplicate segment can be generated when it has been delayed by queueing, etc. If such a segment has the sequence number which falls within the receiver’s current window, the receiver will accept it without any warning or error. However, this segment can be a segment created by an old connection that has the same port and address pair, or a segments sent 2**32 bytes earlier on the same connection. Although this situation rarely happens, it impairs the reliability of TCP.

PAWS utilizes timestamp option in [RFC1323] to provide protection against this. It is assumed that every received TCP segment contains a timestamp. PAWS can identify old duplicate segments by comparing the timestamp in the received segments and the timestamps from other segments received recently. If both TCP endpoints agree to use PAWS, all segments belong to this connection should have timestamp. Since PAWS is the only standardized protection against old duplicate segments, it has been implemented and used in most TCP implementations. However, as some TCP extensions such as [RFC2018], [RFC5925] and [RFC6824] also requires a certain amount of option space in non-SYN segments, using 10-12 bytes length in option space for timestamp in all segments tends to be considered expensive in recent discussions.

In addition, although PAWS is necessary for connections which transmit more than 2**32 bytes, it is not very important for other connections since [RFC0793] already has protection against segments from old connections by using timers. Moreover, some research results indicates that most of TCP flows tend to transmit small amount of data, which means only small fraction of TCP connections really need PAWS [QIAN11]. Timestamp option is also used for RTTM (Round Trip Time Measurement) in [RFC1323]. Gathering many RTT samples from the timestamp in every TCP segment looks useful approach to improve RTO estimation. However, some research results shows the number of samples per RTT does not affect the effectiveness of the RTO [MALLMAN99]. Hence, we can think if PAWS is not used, sending a few timestamps per RTT will be sufficient.

Based on these observations, we propose a new technique called A-PAWS which can archive similar protection against old duplicates segments. The basic idea of A-PAWS is to attain the same protection against old all duplicate segments as PAWS while reducing the use of TS options in segments. A-PAWS is designed to be used complementary with PAWS. This means an implementation that supports A-PAWS is still required to supports PAWS. A-PAWS is activated only when it is safe to use. This sounds the applicability of A-PAWS is limited, however, we believe TCP will have a lot of chances to save the option space if it uses A-PAWS.

There are some discussions that PAWS can also be used to enhance security, however, we still believe that A-PAWS can maintain the same level of security as PAWS. Detailed discussions on this point are provided in Section 5. A-PAWS is an experimental idea yet, but we hope it will contribute to facilitating the use of TCP option space.

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. The A-PAWS Design

A-PAWS assumes PAWS as it is designed to be used complementary with PAWS. Hence, a node which supports A-PAWS MUST support PAWS. The following mechanisms are required in TCP in order to perform A-PAWS.

3.1. Signaling Methods

An endpoint that supports A-PAWS can use the following signaling methods to activate A-PAWS logic.

1) Option Exchange in SYN
   This method uses a new experimental TCP option defined in [RFC6994] and exchanges it during SYN negotiation. The format of the option is depicted in Figure 1. The option does not have any content as it simply indicates the endpoint supports A-PAWS. In this signaling method, when an endpoint wants to use A-PAWS, it MUST put A-PAWS option in SYN or SYN-ACK segment. If an endpoint does not find A-PAWS option in received SYN or SYN-ACK segment, it MUST not send segments with A-PAWS logic in Section 3.3. However, it MUST activate A-PAWS receiver logic in Section 3.4 if it has sent A-PAWS option in SYN or SYN-ACK segment. This is because some middleboxes may remove A-PAWS option in SYN or SYN-ACK segment. A-PAWS receiver logic in Section 3.4 can interact with both A-PAWS and PAWS sender. This signaling requires additional option space in SYN segments, hence non-SYN segment signaling should be used when there is not enough space in SYN option space.

2) Option Exchange in non-SYN Segments
   This method uses the option in Figure 1 as well as the SYN segment signaling. However, the options are not exchanged during SYN negotiation. When a endpoint sets A-PAWS option in the segments, it indicates that it can receive the segments from A-PAWS senders. Hence, it MUST activate A-PAWS receiver logic in Section 3.4 if it sends the options. However, it MUST not send segments with A-PAWS logic in Section 3.3 until it receives A-PAWS options. This approach does not require extra option space or special timestamp value in SYN segments. However, negotiating features in non-SYN segments will require to address further arguments such as when to send the options or how to retransmits the options. We discuss these points in the next section and provide some recommended rules for implementations.

```
                    1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------------------------------+
|  Kind = 254   |  Length = 4   |       16-bit ExID = TBD       |
+---------------+---------------+-------------------------------+
```

Figure 1: A-PAWS option format

3.2. A-PAWS Negotiation Logic for non-SYN Segment Signaling

One important characteristic for A-PAWS is its signaling mechanism does not require tight synchronization between endpoints since A-PAWS receivers can interact with both A-PAWS senders and PAWS senders. This allow us not to invent another three-way handshake like mechanisms for non-SYN segments. This approach will require drastic changes in the current TCP semantics. Instead, we propose a relatively simple and easy mechanism for feature negotiation by using the following rules on A-PAWS endpoints.

Rule 1: An endpoint MUST activate A-PAWS receiver logic in Section 3.4 before it sends A-PAWS option.

Rule 2: An endpoint MUST not send segments with A-PAWS logic in Section 3.3 until it receives A-PAWS option from the other endpoint.

These rules can avoid situations where an endpoint sends segments by A-PAWS logic to an endpoint that doesn't use A-PAWS logic.

Another discussion point for this signaling method is when to set A-PAWS option in segments. As A-PAWS employs asynchronous signaling, both endpoints basically can set A-PAWS option in segments anytime they want. However, it is recommended to use the following rules for setting A-PAWS options.

Rule 3: An endpoint SHOULD use a data segment when it sets A-PAWS option in a segment.

Rule 4: When an endpoint receives a data segment with A-PAWS option, it SHOULD set A-PAWS option for its ACK segment.

Rule 5: An endpoint MAY use A-PAWS options in retransmitted segments.

These rules allow endpoints to have loose synchronized signaling so that they can at least solicit responses from their peers. Of course, even an endpoint solicit a response by setting A-PAWS option in a data segment, it might not receive A-PAWS option in the ACK segment. This can be caused by the lost of the ACK segment or middleboxes that remove unknown options. In order to address these cases, the following rules can be used.

Rule 6: As long as an endpoint does not violate the other rules, it MAY set A-PAWS option in multiple data segments with a certain interval in case no A-PAWS options has been sent from the peer.

This rule can address the cases where A-PAWS options has been removed by middleboxes or segments with A-PAWS options has been lost.

3.3. Sending Behavior

A-PAWS enabled TCP transmits segments, it needs to follow the rules below.

1. TCP needs to check how many bytes has been transmitted in a connection. If the transmitted bytes exceeds 2**32 - 'Sender.Offset', TCP migrates PAWS mode and MUST set timestamp option in all segments to be transmitted. The value for 'Sender.Offset' is discussed in Section 5.

2. If the number of bytes transmitted in a TCP connection does not exceeds 2**32 - 'Sender.Offset', TCP MAY omit timestamp option in segments as long as it does not affect RTTM. This draft does not define how much TCP can omit timestamps because it should be determined by RTTM.

3.4. Receiving Behavior

A-PAWS enabled TCP receives segments, it needs to follow the rules below.

1. TCP needs to check how many bytes has been received in a TCP connection. If it exceeds 2**32 bytes, A-PAWS nodes SHOULD discard the received segments which does not have timestamp option. TCP MUST perform PAWS check when received bytes exceeds 2**32 bytes.

2. If the number of bytes received in a TCP connection does not exceeds 2**32 bytes, A-PAWS nodes SHOULD accept the segments even if it does not have timestamp option. A-PAWS nodes MAY skip PAWS check until the received bytes exceeds 2**32 bytes.

4. When To Activate A-PAWS

In basic principal, A-PAWS capable nodes can always use A-PAWS logic as long as the peers agree with them. However, the following cases require special considerations to enable A-PAWS.

1. As "When To Keep Quiet" section in [RFC0793] suggests, it is recommended that TCP keeps quiet for a MSL upon starting up or recovering from a crash where memory of sequence numbers has been lost. However, if timestamps are being used and if the timestamp clock can be guaranteed to be increased monotonically, this quiet time may be unnecessary. Because TCP can identify the segments

from old connections by checking the timestamp. We think some TCP implementations may disable the quiet time because of using timestamps from this reason. However, since A-PAWS nodes does not set timestamp options in all segments, TCP cannot rely on this approach. To avoid decreasing the robustness of TCP connection, TCP MUST NOT use A-PAWS for a MSL upon starting up or recovering from a crash.

2. Various TCP implementations provide APIs such as setsockopt() that can set SO_REUSEADDR flag on TCP connections. If this flag is set, the TCP connection allows to reuse the same local port without waiting for 2 MSL period. While this option is useful when users want to relaunch applications immediately, it makes the TCP connection a little vulnerable as TCP stack might receive duplicate segments from earlier incarnations. It has been said that PAWS can contribute to mitigate this risk by checking the timestamps in segments. In order to keep the same level of protection, TCP SHOULD NOT send A-PAWS option when SO_REUSEADDR flag is set. This rule prevents the peer from sending segments to this node with A-PAWS logic. However, the node can send segments with A-PAWS logic as long as it received A-PAWS option from the peer.

5. Discussion

As A-PAWS is an experimental logic, the following points need to be considered and discussed.

5.1. Protection Against Early Incarnations

There are some discussions that timestamp can enhance the robustness against early incarnations. Since A-PAWS does not set timestamps in all segments, some may say that it degrades the robustness of TCP. We believe that the degradation caused by A-PAWS on this point is negligible. As long as TCP limits the usage of A-PAWS as described in Section 4, duplicate segments from early incarnations should not be received by TCP.

5.2. Protection Against Security Threats

A TCP connection can be identified by a 5-tuple: source address, destination address, source port number, destination port number and protocol. Crackers need to guess all these parameters when they try malicious attacks on the connection. PAWS can enhance the protection for this as it additionally requires timestamp checking. However, we think the effect of PAWS against malicious attacks is limited due to the simplicity of PAWS check. In PAWS, a segment can be considered as an old duplicate if the timestamp in the segment less than some timestamps recently received on the connection. The "less than" in this context is determined by processing timestamp values as 32 bit unsigned integers in a modular 32-bit space. For example, if t1 and t2 are timestamp values, t1 < t2 is verified when 0 < (t2 - t1) < 2**31 computed in unsigned 32-bit arithmetic. Hence, if crackers set a random value in the timestamp option, there will be 50% chance for them to trick PAWS check. Moreover, there will be more chances if they send multiple segments with different timestamps, which will not be difficult to perform.

In addition, we think there might be a case where using PAWS increases security risks. PAWS recommends to increase timestamp over a system when TCP waives the "quiet time" described in [RFC0793]. However, if timestamps are generated from a global counter, it may leak some information such as system uptime as discussed in [SILBERSACK05]. A-PAWS might be able to allows TCP to use random timestamp values per connections.

5.3. Middlebox Considerations

A-PAWS is designed to be robust against middleboxes. This means that endpoints will not be messed up even if middleboxes discard A-PAWS option. This is because A-PAWS sender logic is activated only when TCP receives a segment with A-PAWS options. A-PAWS receiver logic does not need to know whether the sender is using PAWS or A-PAWS. Activating A-PAWS receiving logic for PAWS sender might be redundant as it requires additional overheads. However, we believe the overhead will be acceptable in most cases because of the simplicity of A-PAWS logic.

Another concern on middleboxes is that they can insert or delete some bytes in TCP connections. If a middlebox inserts extra bytes into a TCP connections, there might be a situation where an A-PAWS sender can transmit segments without timestamp, while an A-PAWS receiver perform PAWS check on them as it already has received 2**32 bytes. In order to avoid discarding segments unnecessarily, we recommend that A-PAWS sender should have a certain amount of offset bytes in order to migrate PAWS mode before the receiver receives 2**32 bytes. We call this protocol parameter 'Sender.Offset'. The proper value for 'Sender.Offset' needs to be discussed.

5.4. Aggressive Mode in A-PAWS

The current A-PAWS requires TCP to migrate PAWS mode after sending/ receiving 2**32 bytes. However, if both nodes check if 2 MSL has already passed during sending/receiving 2**32 bytes, it is safe to continue using A-PAWS. We call this Aggressive mode. The use of Aggressive mode will be explored in future versions.

6. Security Considerations

We believe A-PAWS can maintain the same level of security as PAWS does, but further discussions will be needed. Some security aspects of A-PAWS are discussed in Section 5.

7. IANA Considerations

This document uses the Experimental Option Experiment Identifier. An application for this codepoint in the IANA TCP Experimental Option ExID registry will be submitted.

8. References

8.1. Normative References

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

[RFC1323] Jacobson, V., Braden, R., and D. Borman, "TCP Extensions for High Performance", RFC 1323, DOI 10.17487/RFC1323, May 1992, <http://www.rfc-editor.org/info/rfc1323>.

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

8.2. Informative References

[MALLMAN99] Allman, M. and V. Paxson, "On Estimating End-to-End Network Path Properties", Proceedings of the ACM SIGCOMM , September 1999.

[QIAN11] Qian, L. and B. Carpenter, "A Flow-Based Performance Analysis of TCP and TCP Applications", 3rd International Conference on Computer and Network Technology (ICCNT 2011) , February 2011.

[RFC2018] Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, "TCP Selective Acknowledgment Options", RFC 2018, DOI 10.17487/RFC2018, October 1996, <http://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, <http://www.rfc-editor.org/info/rfc5925>.

[RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <http://www.rfc-editor.org/info/rfc6824>.

[RFC6994] Touch, J., "Shared Use of Experimental TCP Options", RFC 6994, August 2013.

[SILBERSACK05] Silbersack, M., "Improving TCP/IP security through randomization without sacrificing interoperability.", EuroBSDCon 2005 , November 2005.

Author’s Address

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

Email: nishida@wide.ad.jp

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: August 3, 2017

Y. Nishida
GE Global Research
H. Asai
The University of Tokyo
M. Bagnulo
UC3M

January 30, 2017

Increasing Maximum Window Size of TCP
draft-nishida-tcpm-maxwin-03.txt

Abstract

This document proposes to increase the current max window size allowed in TCP. It describes the current logic that limits the maximum window size and provides a rationale to relax the limitation as well as the negotiation mechanism to enable this feature safely.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

TCP throughput is determined by two factors: Round Trip Time and Receive Window size. It can never exceed Receive Window size divided by RTT. This implies larger window size is important to achieve better performance. Original TCP's maximum window size defined in RFC793 [RFC0793] is 2^16 -1 (65,535), however, RFC7323 [RFC7323] defines TCP Window Scale option which allows TCP to use larger window size. Window Scale uses a shift count stored in 1-byte field in the option. The receiver of the option uses left-shifted window size value by the shift count as actual window size. When Window Scale is used, TCP can extend maximum window size to 2^30 - 2^14 (1,073,725,440). This is because the maximum shift count is 14 as described in the Section 2.3 of RFC7323 [RFC7323]. However, since TCP's sequence number space is 2^32, we believe it is still possible to use larger window size than this while careful design of the logic that can identify segments inside the window is required. In this document, we propose to increase the maximum shift count to 15, which extend window size to 2^31 - 2^15.

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. Increasing Maximum Window Size

RFC7323 requires maximum window size to be less than 2^30 as described below.

> "
> TCP determines if a data segment is "old" or "new" by testing whether its sequence number is within 2^31 bytes of the left edge of the window, and if it is not, discarding the data as "old". To insure that new data is never mistakenly considered old and vice versa, the left edge of the sender's window has to be at most 2^31 away from the right edge of the receiver's window. The same is true of the sender's right edge and receiver's left edge. Since the right and left edges of either the sender's or receiver's window differ by the window size, and since the sender and receiver windows can be out of phase by at most the window size, the above constraints imply that two times the maximum window size must be less than 2^31, or
>
> max window < 2^30
> "

However, TCP does not necessarily need to determine if a segment is old or new. Because important point is to determine if a receive segment is inside of the window or not. It basically does not matter if a segment is too old (left side of the window) or too new (right side of the window) as long as it is outside of the window. Based on this viewpoint, we propose to extend maximum window to 2^31 - 2^15, which can be attained by increasing maximum shift count to 15.

To demonstrate the feasibility of the proposal, we would like to use the following worst case example where the sender and the receiver windows are completely out of phase. In this example, we define S as the sender's left edge of the window and W as the sender's window size. Hence, the sender's right edge of the window is S+W. Also, the receiver's left edge of the window is S+W+1 and the right edge of the window is S+2W+1, as they are out of phase. This situation can happen when the sender sent all segments in the window and the receiver received all segments while no ACK has been received by the sender yet. Now, we presume a segment that contains sequence number S has arrived at the receiver. This segment should be excluded by the receiver, although it can easily happen when the sender retransmits segments.

In case of W=2^31, the receiver cannot exclude this segment as S+2W = S. It is considered inside of the window. (S+W+1 < S < S+2W+1) However, our proposed window size is W=2^31-X, where X is 2^15. In this case, when segment S has arrived, the following checks will be performed. First, TCP checks it with the left edge of the window and it considers the segment is left side of the left edge. (S < S+W+1 Note: W=2^31-X) Second, TCP checks it with the right edge of the window and it considers the segment is right of the right edge. (S > S+2W+1) You might notice that the result of the second check is not expected one as the segment S is actually an old segment. This is the problem that the referred paragraphs from RFC7323 [RFC7323] describe. However, the segment is properly excluded by the receiver as both checks indicate it is outside of the window. It should be noted that the principle of TCP requires to accept the segment S only when it has passed both checks successfully, which means S must satisfy the following condition.

S >= left edge && S <= right edge

As we have shown in the example, our proposed maximum window size: W=2^31-2^15 does not affect this principle.

Using the larger window size implies that the sequence number space can wrap around in less than 3 RTTs. This can pose problems to distinguish old retransmitted packets from new packets solely using the same sequence number. Because of this, a sender using the larger window size defined in this specification is recommended to use Protection Against Wrapped Sequences (PAWS) as defined in RFC7323 [RFC7323].

4. Updating the Window Scale Option

As shown in Figure 1, the Window Scale Option (WSO) defined in [RFC7323] has three 1-byte fields, the Kind field (which specifies the option type), the Length field (set to 3 because the WSO is 3 bytes long) and the shift.cnt field (which specifies the shift count applied to the window to scale it).

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

Figure 1: Window Scale Option (WSO) format

RFC7323 [RFC7323] defines that the shift.cnt field can have a maximum value of 14 and upon reception of a larger value in this field, the receiver must proceed as if it had received a shift.cnt of 14.

This specification updates the shift.cnt field definition. Figure 2 represents the new format of the shift.cnt field. The eight bits contained in the shift.cnt field are formatted as "SSSSLRRR".

```
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|S S S S L R R R|
+-+-+-+-+-+-+-+-+
```

Figure 2: New shift.cnt field format

These bits are parsed as follows:

- o The four leftmost bits "SSSS" express the shift-count, as in RFC7323 [RFC7323], only that now the maximum shift count value allowed is 15.

- o The "L" bit expresses if the sender supports the large window defined in this specification i.e. the bit is set if the sender supports this specification.

- o The three rightmost bits "RRR" are reserved for future use and MUST be set to zero.

This new format for the shift.count field allows an updated client to initiate a TCP connection and express that it supports the larger window by setting the "L" bit, while still conveying information about the shift count that it wants to use for its own RCV.WND in the four leftmost bits "SSSS" (which do not necessarily have to be set to 15). A server that supports this specification that receives a SYN with the WSO with the "L" bit set knows that it can reply using a shift count of 15. A legacy server that receives the WSO with the "L"" bit set will interpret it using the RFC7323 format and will then read it as a shift count value larger than 14. As per RFC7323 the server MUST then assume a shift count of 14. The legacy server will then reply with a WSO with the "L" bit set to zero, so the client knows that the server does not support this specification and that the server will assume a shift count of 14 for the client’s receive window.

5. Use Cases, Benefits to Explore Maximum Window Size

One of the use cases of the extended maximum window size is high volume data transfer over paths with long RTT delays and high bandwidth, called long fat pipes. The proposed extension improves and doubles at most the maximum throughput when bandwidth-latency product is greater than 1 GB. As propagation delay in an optical fiber is around 20 cm/ns, RTT will be over 100 milliseconds when the distance of the transmission is more than 10000km. This distance is not extraordinary for trans-pacific communications. In this case, the maximum throughput will be limited to 80 Gbps with the current maximum window size, although network technologies for more than 100 Gbps are becoming common these days.

As the current TCP sequence number space is limited to 32 bits, it will not be possible to increase maximum window size any further. However, TCP may eventually have other extensions to increase sequence number space, for example, [RFC7323] and [RFC1263] mention about increasing sequence number space to 64 bits. We believe the information in this document will be useful when such extensions are proposed as they need to define new maximum window size.

6. Acknowledgments

The authors gratefully acknowledge significant inputs for this document from Richard Scheffenegger and Ilpo Jarvinen.

7. Security Considerations

It is known that an attacker can have more chances to insert forged packets into a TCP connection when large window size is used. This is not a specific problem of this proposal, but a generic problem to use larger window. Using PAWS can mitigate this problem, however, it is recommended to consult the Security Considerations section of RFC7323 [RFC7323] to check its implications.

8. IANA Considerations

If approved, this document overrides the definition of the WSO option defined in RFC7323 and so the IANA registry should be update accordingly (at least to add a pointer to this specification as reference for the WSO in the IANA registry).

9. References

9.1. Normative References

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

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

[RFC7323] Borman, D., Braden, B., Jacobson, V., and R. Scheffenegger, Ed., "TCP Extensions for High Performance", RFC 7323, DOI 10.17487/RFC7323, September 2014, <http://www.rfc-editor.org/info/rfc7323>.

9.2. Informative References

[RFC1263] O'Malley, S. and L. Peterson, "TCP Extensions Considered Harmful", RFC 1263, DOI 10.17487/RFC1263, October 1991, <http://www.rfc-editor.org/info/rfc1263>.

Authors' Addresses

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

Email: nishida@wide.ad.jp

Hirochika Asai
The University of Tokyo
7-3-1 Hongo
Bunkyo-ku, Tokyo 113-8656
JP

Email: panda@wide.ad.jp

Marcelo Bagnulo
UC3M

Email: marcelo@it.uc3m.es

TCP Maintenance and Minor Extensions (tcpm) M. Welzl
Internet-Draft S. Islam
Intended status: Informational University of Oslo
Expires: April 21, 2016 J. Touch
USC/ISI
J. You
Huawei
October 19, 2015

The state of implementation of TCP control block interdependence
draft-welzl-tcpm-tcb-sharing-00

Abstract

This document provides an overview of the state of implementation of RFC 2140, in preparation for a possible future RFC2140bis document.

Table of Contents

1. State of Implementation

* L = Linux, F = FreeBSD

Table 1: State of implementation of RFC 2140 in Linux and FreeBSD

RFC 2140	Description	Implementation	Status
Old-MSS	Maximum Segment Size	F:rmx_mtu	This is being cached and shared in FreeBSD.
Old-RTT	Estimated Round-Trip Time	L:TCP_METRIC_RTT F:rmx_rtt	Cached in both FreeBSD and Linux, however it is being used by a new connection in FreeBSD only.
Old-RTT var	Estimated Round-Trip Time	L:TCP_METRIC_ RTTVAR F:rmx_rttvar	Cached in both FreeBSD and Linux, however it is being used by a new connection in FreeBSD only.
Old-snd_ cwnd	Congestion Window	L:TCP_METRIC_ CWND F:rmx_cwnd	Cached in both FreeBSD and Linux, however it is not being used by a new connection.
-	Slow Start Thresold	L:TCP_METRIC_ SSTHRESH F:rmx_ssthresh	This is being cached and shared in both FreeBSD and Linux. In Linux, it is set to max(cwnd/2, ssthresh) in most cases. In

			FreeBSD, however, it is set to either the current ssthresh if not set previously, or to the arithmetic ssthresh and previously cached metric.
-	Metric related to the extent of reordering.	L:TCP_METRIC_ REORDERING	This is being cached and shared in Linux.
-	Estimated Bandwidth	F:rmx_bandwidth	Not in the specification. It is not set before caching when a connection is closed.
-	Outbound Delay - Bandwidth Product	F:rmx_sendpipe	Not in the specification. This is used for socket buffer in FreeBSD. The value is set to 0 before caching when a connection is closed.
-	Inbound Delay- Bandwidth Product	F:rmx_recvpipe	Not in the specification. This is used for socket buffer in FreeBSD. The value is set to 0 before caching when a connection is closed.

2. IANA Considerations

This memo includes no request to IANA.

3. Security Considerations

To be added

Authors' Addresses

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

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

Safiqul Islam
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Phone: +47 22 84 08 37
Email: safiquli@ifi.uio.no

Joe Touch
USC/ISI
4676 Admiralty Way, Marina del Rey
CA 90292-6695
USA

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

Jianjie You
Huawei
101 Software Avenue, Yuhua District
Nanjing 210012
China

Email: youjianjie@huawei.com