

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: April 23, 2018

N. Khademi
M. Welzl
University of Oslo
G. Armitage
Swinburne University of Technology
G. Fairhurst
University of Aberdeen
October 20, 2017

TCP Alternative Backoff with ECN (ABE)
draft-ietf-tcpm-alternativebackoff-ecn-02

Abstract

Recent Active Queue Management (AQM) mechanisms instantiate shallow buffers with burst tolerance to minimise the time that packets spend enqueued at a bottleneck. However, shallow buffering can cause noticeable performance degradation when TCP is used over a network path with a large bandwidth-delay-product. Traditional methods rely on detecting network congestion through reported loss of transport packets. Explicit Congestion Notification (ECN) instead allows a router to directly signal incipient congestion. A sending endpoint can distinguish when congestion is signalled via ECN, rather than by packet loss. An ECN signal indicates that an AQM mechanism has done its job, and therefore the bottleneck network queue is likely to be shallow. This document therefore proposes an update to the TCP sender-side ECN reaction in congestion avoidance to reduce the Congestion Window (cwnd) by a smaller amount than the congestion control algorithm's reaction to loss. This document also recommends this approach to be adopted by any other transport protocol that implements a congestion control reduction to an ECN congestion signal.

Status of This Memo

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

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

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

This Internet-Draft will expire on April 23, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Definitions	2
2. Introduction	2
3. Specification	4
4. Discussion	4
4.1. Why Use ECN to Vary the Degree of Backoff?	4
4.2. Focus on ECN as Defined in RFC3168	5
4.3. Discussion: Choice of ABE Multiplier	5
5. Status of the Update	7
6. Acknowledgements	7
7. IANA Considerations	8
8. Implementation Status	8
9. Security Considerations	8
10. Revision Information	8
11. References	9
11.1. Normative References	9
11.2. Informative References	10
Authors' Addresses	11

1. Definitions

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

2. Introduction

Explicit Congestion Notification (ECN) [RFC3168] makes it possible for an Active Queue Management (AQM) mechanism to signal the presence of incipient congestion without incurring packet loss. This lets the

network deliver some packets to an application that would have been dropped if the application or transport did not support ECN. This packet loss reduction is the most obvious benefit of ECN, but it is often relatively modest. There are also significant other benefits from deploying ECN [RFC8087], including reduced end-to-end network latency.

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

Research has demonstrated the benefits of reducing network delays due to excessive buffering [BUFFERBLOAT]; this has led to the creation of new AQM mechanisms like PIE [RFC8033] and CoDel [CODEL2012] [I-D.CoDel], which avoid causing bloated queues that are common with a simple tail-drop behaviour (also known as a First-In First-Out, FIFO, queue).

These AQM mechanisms instantiate short queues that are designed to tolerate packet bursts. However, congestion control mechanisms cannot always utilise a bottleneck link well where there are short queues. For example, to allow a single TCP connection to fully utilise a network path, the queue at the bottleneck link must be able to compensate for TCP halving the "cwnd" and "ssthresh" variables in response to a lost packet [RFC5681]. This requires the bottleneck queue to be able to store at least an end-to-end bandwidth-delay product (BDP) of data, which effectively doubles both the amount of data that can be in flight and the round-trip time (RTT) experience using the network path.

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

3. Specification

This specification describes an update to the congestion control algorithm of an ECN-capable TCP transport protocol. It allows a TCP stack to update the TCP sender response when it receives feedback indicating reception of a CE-marked packet. It RECOMMENDS that a TCP sender multiplies the cwnd by 0.8 and reduces the slow start threshold (ssthresh) in congestion avoidance following reception of a TCP segment that sets the ECN-Echo flag (defined in [RFC3168]). While this specification concerns TCP, other transports also support a per-RTT response to ECN. The method defined in this document is also applicable for such transports.

4. Discussion

Much of the technical background to this congestion control response can be found in a research paper [ABE2017]. This paper used a mix of experiments, theory and simulations with standard NewReno and CUBIC to evaluate the technique. It examined the impact of enabling ECN and letting individual TCP senders back off by a reduced amount in reaction to the receiver that reports ECN CE-marks from AQM-enabled bottlenecks. The technique was shown to present "...significant performance gains in lightly-multiplexed scenarios, without losing the delay-reduction benefits of deploying CoDel or PIE". The performance improvement is achieved when reacting to ECN-Echo in congestion avoidance by multiplying cwnd and ssthresh with a value in the range [0.7..0.85].

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

The classic rule-of-thumb dictates that a network path needs to provide a BDP of bottleneck buffering if a TCP connection wishes to optimise path utilisation. A single TCP bulk transfer running through such a bottleneck will have increased its congestion window (cwnd) up to $2 \times \text{BDP}$ by the time that packet loss occurs. When packet loss is detected (regarded as a notification of congestion), Standard TCP halves the cwnd and ssthresh [RFC5681], which causes the TCP congestion control to go back to allowing only a BDP of packets in flight -- just sufficient to maintain 100% utilisation of the bottleneck on the network path.

AQM mechanisms such as CoDel [I-D.CoDel] and PIE [RFC8033] set a delay target in routers and use congestion notifications to constrain the queuing delays experienced by packets, rather than in response to impending or actual bottleneck buffer exhaustion. With current default delay targets, CoDel and PIE both effectively emulate a shallow buffered bottleneck (section II, [ABE2017]) while also allowing short traffic bursts into the queue. This provides

acceptable performance for TCP connections over a path with a low BDP, or in highly multiplexed scenarios (many concurrent transport connections). However, it interacts badly for a lightly-multiplexed case (few concurrent connections) over a path with a large BDP. Conventional TCP backoff in such cases leads to gaps in packet transmission and under-utilisation of the path.

Instead of discarding packets, an AQM mechanism is allowed to mark ECN-capable packets with an ECN CE-mark. The reception of a CE-mark not only indicates congestion on the network path, it also indicates that an AQM mechanism exists at the bottleneck along the path, and hence the CE-mark likely came from a bottleneck with a shallow queue. Reacting differently to an ECN CE-mark than to packet loss can then yield the benefit of a reduced back-off, as with CUBIC [I-D.CUBIC], when queues are short, yet it can avoid generating excessive delay when queues are long. Using ECN can also be advantageous for several other reasons [RFC8087].

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

4.2. Focus on ECN as Defined in RFC3168

Some transport protocol mechanisms rely on ECN semantics that differ from the original ECN definition [RFC3168] -- for example, Congestion Exposure (ConEx) [RFC7713] and Datacenter TCP (DCTCP) [I-D.ietf-tcpm-dctcp] need more accurate ECN information than that offered by the original feedback method. Other mechanisms (e.g., [I-D.ietf-tcpm-accurate-ecn]) allow the sender to adjust the rate more frequently than once each path RTT. Use of these mechanisms is out of the scope of the current document.

4.3. Discussion: Choice of ABE Multiplier

ABE decouples the reaction of a TCP sender to loss and ECN CE-marks when in the congestion avoidance phase by differentiating the scaling factor used in Equation 4 in Section 3.1 of [RFC5681]. The description respectively uses β_{loss} and β_{ecn} to refer to the multiplicative decrease factors applied in response to packet loss, and in response to a receiver indicating that an ECN CE-mark was received on an ECN-enabled TCP connection. For non-ECN-enabled

TCP connections, no ECN CE-marks are received and only β_{loss} applies.

In other words, in response to detected loss:

$$\text{ssthresh}_{(t+1)} = \max(\text{FlightSize}_t * \beta_{\text{loss}}, 2 * \text{SMSS})$$

and in response to an indication of a received ECN CE-mark:

$$\text{ssthresh}_{(t+1)} = \max(\text{FlightSize}_t * \beta_{\text{ecn}}, 2 * \text{SMSS})$$

and

$$\text{cwnd}_{(t+1)} = \text{ssthresh}_{(t+1)}$$

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

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

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

β_{ecn} depends on how the response of a TCP connection to shallow AQM marking thresholds is optimised. β_{loss} reflects the preferred response of each congestion control algorithm when faced with exhaustion of buffers (of unknown depth) signalled by packet loss. Consequently, for any given TCP congestion control algorithm the choice of β_{ecn} is likely to be algorithm-specific, rather than a constant multiple of the algorithm's existing β_{loss} .

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

5. 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. It does not require any ABE-specific changes in routers or the use of Accurate ECN feedback [I-D.ietf-tcpm-accurate-ecn] by a receiver.

The currently published ECN specification requires that the congestion control response to a CE-marked packet is the same as the response to a dropped packet [RFC3168]. The specification is currently being updated to allow for specifications that do not follow this rule [I-D.ECN-exp]. The present specification defines such an experiment and has thus been assigned an Experimental status before being proposed as a Standards-Track update.

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

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

To evaluate the benefit, this experiment therefore requires support in AQM routers (except to enable an ECN-marking mechanism [RFC3168] [RFC7567]) for ECN-marking of packets carrying the ECN Capable Transport, ECT(0), codepoint [RFC3168].

If the method is only deployed by some senders, and not by others, the senders that use this method can gain some advantage, possibly at the expense of other flows that do not use this updated method. Because this advantage applies only to ECN-marked packets and not to loss indications, the new method cannot lead to congestion collapse.

The result of this Internet experiment will be reported by presentation to the TCPM WG (or IESG) or an implementation report at the end of the experiment.

6. Acknowledgements

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

The authors would like to thank Stuart Cheshire for many suggestions when revising the draft, and the following people for their contributions to [ABE2017]: Chamil Kulatunga, David Ros, Stein

Gjessing, Sebastian Zander. Thanks also to (in alphabetical order) Bob Briscoe, Markku Kojo, John Leslie, Dave Taht and the TCPM working group for providing valuable feedback on this document.

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

7. IANA Considerations

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

This document includes no request to IANA.

8. Implementation Status

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

9. Security Considerations

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

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

10. Revision Information

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

-02. Corrected the equations in Section 4.3. Updated the affiliations. Lower bound for cwnd is defined. A recommendation for window-based transport protocols is changed to cover all transport

protocols that implements a congestion control reduction to an ECN congestion signal. Added text about ABE's FreeBSD mainline kernel status including a reference to the FreeBSD code review page. References are updated.

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

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

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

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

11. References

11.1. Normative References

[I-D.ECN-exp]

Black, D., "Explicit Congestion Notification (ECN) Experimentation", Internet-draft, IETF work-in-progress draft-ietf-tsvwg-ecn-experimentation-06, September 2017.

[RFC2119]

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

[RFC3168]

Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <<https://www.rfc-editor.org/info/rfc3168>>.

- [RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <<https://www.rfc-editor.org/info/rfc5681>>.
- [RFC7567] Baker, F., Ed. and G. Fairhurst, Ed., "IETF Recommendations Regarding Active Queue Management", BCP 197, RFC 7567, DOI 10.17487/RFC7567, July 2015, <<https://www.rfc-editor.org/info/rfc7567>>.

11.2. Informative References

- [ABE-FreeBSD] "ABE patch review in FreeBSD", <<https://reviews.freebsd.org/D11616>>.
- [ABE2017] Khademi, N., Armitage, G., Welzl, M., Fairhurst, G., Zander, S., and D. Ros, "Alternative Backoff: Achieving Low Latency and High Throughput with ECN and AQM", IFIP NETWORKING 2017, Stockholm, Sweden, June 2017.
- [BUFFERBLOAT] "Bufferbloat project", <<https://www.bufferbloat.net/projects/bloat/wiki/Introduction/>>.
- [CODEL2012] Nichols, K. and V. Jacobson, "Controlling Queue Delay", July 2012, <<http://queue.acm.org/detail.cfm?id=2209336>>.
- [I-D.CoDel] Nichols, K., Jacobson, V., McGregor, V., and J. Iyengar, "Controlled Delay Active Queue Management", Internet-draft, IETF work-in-progress draft-ietf-aqm-codel-09, September 2017.
- [I-D.CUBIC] Rhee, I., Xu, L., Ha, S., Zimmermann, A., Eggert, L., and R. Scheffenegger, "CUBIC for Fast Long-Distance Networks", Internet-draft, IETF work-in-progress draft-ietf-tcpm-cubic-06, September 2017.
- [I-D.ietf-tcpm-accurate-ecn] Briscoe, B., Kuehlewind, M., and R. Scheffenegger, "More Accurate ECN Feedback in TCP", draft-ietf-tcpm-accurate-ecn-03 (work in progress), May 2017.

- [I-D.ietf-tcpm-dctcp]
Bensley, S., Thaler, D., Balasubramanian, P., Eggert, L.,
and G. Judd, "Datacenter TCP (DCTCP): TCP Congestion
Control for Datacenters", draft-ietf-tcpm-dctcp-10 (work
in progress), August 2017.
- [ICC2002] Kwon, M. and S. Fahmy, "TCP Increase/Decrease Behavior
with Explicit Congestion Notification (ECN)", IEEE
ICC 2002, New York, New York, USA, May 2002,
<<http://dx.doi.org/10.1109/ICC.2002.997262>>.
- [RFC7713] Mathis, M. and B. Briscoe, "Congestion Exposure (ConEx)
Concepts, Abstract Mechanism, and Requirements", RFC 7713,
DOI 10.17487/RFC7713, December 2015,
<<https://www.rfc-editor.org/info/rfc7713>>.
- [RFC8033] Pan, R., Natarajan, P., Baker, F., and G. White,
"Proportional Integral Controller Enhanced (PIE): A
Lightweight Control Scheme to Address the Bufferbloat
Problem", RFC 8033, DOI 10.17487/RFC8033, February 2017,
<<https://www.rfc-editor.org/info/rfc8033>>.
- [RFC8087] Fairhurst, G. and M. Welzl, "The Benefits of Using
Explicit Congestion Notification (ECN)", RFC 8087,
DOI 10.17487/RFC8087, March 2017,
<<https://www.rfc-editor.org/info/rfc8087>>.

Authors' Addresses

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

Email: naeemk@ifi.uio.no

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

Email: michawe@ifi.uio.no

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

Email: garmitage@swin.edu.au

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

Email: gorry@erg.abdn.ac.uk

TCP Maintenance Working Group
Internet-Draft
Intended status: Experimental
Expires: September 14, 2017

Y. Cheng
N. Cardwell
N. Dukkipati
Google, Inc
March 13, 2017

RACK: a time-based fast loss detection algorithm for TCP
draft-ietf-tcpm-rack-02

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on September 14, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

1. Introduction

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 N² interactions is a daunting task and error-prone.

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

2. Overview

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

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

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

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

3. Requirements

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

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

RACK has three requirements:

1. The connection MUST use selective acknowledgment (SACK) options [RFC2018].
2. For each packet sent, the sender MUST store its most recent transmission time with (at least) millisecond granularity. For round-trip times lower than a millisecond (e.g., intra-datacenter communications) microsecond granularity would significantly help the detection latency but is not required.
3. For each packet sent, the sender MUST remember whether the packet has been retransmitted or not.

We assume that requirement 1 implies the sender keeps a SACK scoreboard, which is a data structure to store selective acknowledgment information on a per-connection basis ([RFC6675] section 3). For the ease of explaining the algorithm, we use a pseudo-scoreboard that manages the data in sequence number ranges. But the specifics of the data structure are left to the implementor.

RACK does not need any change on the receiver.

4. Definitions of variables

A sender needs to store these new RACK variables:

"Packet.xmit_ts" is the time of the last transmission of a data packet, including retransmissions, if any. The sender needs to record the transmission time for each packet sent and not yet acknowledged. The time MUST be stored at millisecond granularity or finer.

"RACK.packet". Among all the packets that have been either selectively or cumulatively acknowledged, RACK.packet is the one that was sent most recently (including retransmissions).

"RACK.xmit_ts" is the latest transmission timestamp of RACK.packet.

"RACK.end_seq" is the ending TCP sequence number of RACK.packet.

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

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

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

"RACK.ack_ts" is the time when all the sequences in RACK.packet were selectively or cumulatively acknowledged.

Note that the Packet.xmit_ts variable is per packet in flight. The RACK.xmit_ts, RACK.end_seq, RACK.RTT, RACK.reo_wnd, and RACK.min_RTT variables are kept in the per-connection TCP control block. RACK.packet and RACK.ack_ts are used as local variables in the algorithm.

5. Algorithm Details

5.1. Transmitting a data packet

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

5.2. Upon receiving an ACK

Step 1: Update RACK.min_RTT.

Use the RTT measurements obtained 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. [REORDER-DETECT][RFC4737] MAY be implemented to dynamically adjust the reordering window upon detecting reordering. 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 have been retransmitted before and either of two conditions is true:

1. The Timestamp Echo Reply field (TSecr) of the ACK's timestamp option [RFC7323], if available, indicates the ACK was not acknowledging the last retransmission of the packet.
2. The packet was last retransmitted less than RACK.min_rtt ago. While it is still possible the packet is spuriously retransmitted because of a recent RTT decrease, we believe that our experience suggests this is a reasonable heuristic.

If 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, then the packet was likely lost.

If another packet that was sent later has been delivered, but the reordering window has not passed, then it is not yet safe to deem the unacked packet lost. Using the basic algorithm above, the sender would wait for the next ACK to further advance RACK.xmit_ts; but this risks a timeout (RTO) if no more ACKs come back (e.g, due to losses or application limit). For timely loss detection, the sender MAY install a "reordering settling" timer set to fire at the earliest moment at which it is safe to conclude that some packet is lost. The earliest moment is the time it takes to expire the reordering window of the earliest unacked packet in flight.

This timer expiration value can be derived as follows. As a starting point, we consider that the reordering window has passed if the RACK.packet was sent sufficiently after the packet in question, or a

sufficient time has elapsed since the RACK.packet was S/ACKed, or some combination of the two. More precisely, RACK marks a packet as lost if the reordering window for a packet has elapsed through the sum of:

1. delta in transmit time between a packet and the RACK.packet
2. delta in time between RACK.ack_ts and now

So we mark a packet as lost if:

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

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

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

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

```
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
  /* Timestamp tie breaker */
  If Packet.xmit_ts == RACK.xmit_ts AND
  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. Tail Loss Probe: fast recovery on tail losses

This section describes a supplemental algorithm, Tail Loss Probe (TLP), which leverages RACK to further reduce RTO recoveries. TLP triggers fast recovery to quickly repair tail losses that can otherwise be recovered by RTOs only. After an original data transmission, TLP sends a probe data segment within one to two RTTs. The probe data segment can either be new, previously unsent data, or a retransmission of previously sent data just below SND.NXT. In either case the goal is to elicit more feedback from the receiver, in the form of an ACK (potentially with SACK blocks), to allow RACK to trigger fast recovery instead of an RTO.

An RTO occurs when the first unacknowledged sequence number is not acknowledged after a conservative period of time has elapsed [RFC6298]. Common causes of RTOs include:

1. Tail losses at the end of an application transaction.
2. Lost retransmits, which can halt fast recovery based on [RFC6675] if the ACK stream completely dries up. For example, consider a window of three data packets (P1, P2, P3) that are sent; P1 and P2 are dropped. On receipt of a SACK for P3, RACK marks P1 and P2 as lost and retransmits them as R1 and R2. Suppose R1 and R2

are lost as well, so there are no more returning ACKs to detect R1 and R2 as lost. Recovery stalls.

3. Tail losses of ACKs.
4. An unexpectedly long round-trip time (RTT). This can cause ACKs to arrive after the RTO timer expires. The F-RTO algorithm [RFC5682] is designed to detect such spurious retransmission timeouts and at least partially undo the consequences of such events, but F-RTO cannot be used in many situations.

6.1. Tail Loss Probe: An Example

Following is an example of TLP. All events listed are at a TCP sender.

(1) Sender transmits segments 1-10: 1, 2, 3, ..., 8, 9, 10. There is no more new data to transmit. A PTO is scheduled to fire in 2 RTTs, after the transmission of the 10th segment. (2) Sender receives acknowledgements (ACKs) for segments 1-5; segments 6-10 are lost and no ACKs are received. The sender reschedules its PTO timer relative to the last received ACK, which is the ACK for segment 5 in this case. The sender sets the PTO interval using the calculation described in step (2) of the algorithm. (3) When PTO fires, sender retransmits segment 10. (4) After an RTT, a SACK for packet 10 arrives. The ACK also carries SACK holes for segments 6, 7, 8 and 9. This triggers RACK-based loss recovery. (5) The connection enters fast recovery and retransmits the remaining lost segments.

6.2. Tail Loss Probe Algorithm Details

We define the terminology used in specifying the TLP algorithm:

FlightSize: amount of outstanding data in the network, as defined in [RFC5681].

RTO: The transport's retransmission timeout (RTO) is based on measured round-trip times (RTT) between the sender and receiver, as specified in [RFC6298] for TCP. PTO: Probe timeout (PTO) is a timer event indicating that an ACK is overdue. Its value is constrained to be smaller than or equal to an RTO.

SRTT: smoothed round-trip time, computed as specified in [RFC6298].

Open state: the sender has so far received in-sequence ACKs with no SACK blocks, and no other indications (such as retransmission timeout) that a loss may have occurred.

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

6.2.1. Phase 1: Scheduling a loss probe

Step 1: Check conditions for scheduling a PTO.

A sender should schedule a PTO after transmitting new data or receiving an ACK if the following conditions are met:

- (a) The connection is in Open state
- (b) The connection is either cwnd-limited (the data in flight matches or exceeds the cwnd) or application-limited (there is no unsent data that the receiver window allows to be sent)
- (c) SACK is enabled for the connection
- (d) The most recently transmitted data was not itself a TLP probe (i.e. a sender MUST NOT send consecutive TLP probes)

Step 2: Select the duration of the PTO.

A sender SHOULD use the following logic to select the duration of a PTO:

```
If an SRTT estimate is available:
    PTO = 2 * SRTT
Else:
    PTO = initial RTO of 1 sec
If FlightSize = 1:
    PTO = max(PTO, 1.5 * SRTT + WCDelAckT)
PTO = max(10ms, PTO)
PTO = max(RTO, PTO)
```

Aiming for a PTO value of $2 \times \text{SRTT}$ allows a sender to wait long enough to know that an ACK is overdue. Under normal circumstances, i.e. no losses, an ACK typically arrives in one SRTT. But choosing PTO to be exactly an SRTT is likely to generate spurious probes given that network delay variance and even end-system timings can easily push an ACK to be above an SRTT. We chose PTO to be the next integral multiple of SRTT. Similarly, current end-system processing latencies and timer granularities can easily push an ACK beyond 10ms, so senders SHOULD use a minimum PTO value of 10ms. If RTO is smaller

than the computed value for PTO, then a probe is scheduled to be sent at the RTO time.

WCDelAckT stands for worst case delayed ACK timer. When FlightSize is 1, PTO is inflated additionally by WCDelAckT time to compensate for a potential long delayed ACK timer at the receiver. The RECOMMENDED value for WCDelAckT is 200ms, or the delayed ACK interval value explicitly negotiated by the sender and receiver, if one is available.

6.2.2. Phase 2: Sending a loss probe

When the PTO fires, transmit a probe data segment:

```
If a previously unsent segment exists AND
the receive window allows new data to be sent:
    Transmit that new segment
    FlightSize += SMSS
Else:
    Retransmit the last segment
The cwnd remains unchanged
```

6.2.3. Phase 3: ACK processing

On each incoming ACK, the sender should cancel any existing loss probe timer. The sender should then reschedule the loss probe timer if the conditions in Step 1 of Phase 1 allow.

6.3. TLP recovery detection

If the only loss in an outstanding window of data was the last segment, then a TLP loss probe retransmission of that data segment might repair the loss. TLP recovery detection examines ACKs to detect when the probe might have repaired a loss, and thus allows congestion control to properly reduce the congestion window (cwnd) [RFC5681].

Consider a TLP retransmission episode where a sender retransmits a tail packet in a flight. The TLP retransmission episode ends when the sender receives an ACK with a SEG.ACK above the SND.NXT at the time the episode started. During the TLP retransmission episode the sender checks for a duplicate ACK or D-SACK indicating that both the original segment and TLP retransmission arrived at the receiver, meaning there was no loss that needed repairing. If the TLP sender does not receive such an indication before the end of the TLP retransmission episode, then it MUST estimate that either the original data segment or the TLP retransmission were lost, and

congestion control MUST react appropriately to that loss as it would any other loss.

Since a significant fraction of the hosts that support SACK do not support duplicate selective acknowledgments (D-SACKs) [RFC2883] the TLP algorithm for detecting such lost segments relies only on basic SACK support [RFC2018].

Definitions of variables

TLPRxtOut: a boolean indicating whether there is an unacknowledged TLP retransmission.

TLPHighRxt: the value of SND.NXT at the time of sending a TLP retransmission.

6.3.1. Initializing and resetting state

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

```
TLPRxtOut = false
```

6.3.2. Recording loss probe states

Senders must only send a TLP loss probe retransmission if TLPRxtOut is false. This ensures that at any given time a connection has at most one outstanding TLP retransmission. This allows the sender to use the algorithm described in this section to estimate whether any data segments were lost.

Note that this condition only restricts TLP loss probes that are retransmissions. There may be an arbitrary number of outstanding unacknowledged TLP loss probes that consist of new, previously-unsent data, since the retransmission timeout and fast recovery algorithms are sufficient to detect losses of such probe segments.

Upon sending a TLP probe that is a retransmission, the sender sets TLPRxtOut to true and TLPHighRxt to SND.NXT.

Detecting recoveries accomplished by loss probes

Step 1: Track ACKs indicating receipt of original and retransmitted segments

A sender considers both the original segment and TLP probe retransmission segment as acknowledged if either (i) or (ii) are true:

(i) This is a duplicate acknowledgment (as defined in [RFC5681], section 2), and all of the following conditions are met:

- (a) TLPRxtOut is true
- (b) SEG.ACK == TLPHighRxt
- (c) SEG.ACK == SND.UNA
- (d) the segment contains no SACK blocks for sequence ranges above TLPHighRxt
- (e) the segment contains no data
- (f) the segment is not a window update

(ii) This is an ACK acknowledging a sequence number at or above TLPHighRxt and it contains a D-SACK; i.e. all of the following conditions are met:

- (a) TLPRxtOut is true
- (b) SEG.ACK >= TLPHighRxt and
- (c) the ACK contains a D-SACK block

If either conditions (i) or (ii) are met, then the sender estimates that the receiver received both the original data segment and the TLP probe retransmission, and so the sender considers the TLP episode to be done, and records that fact by setting TLPRxtOut to false.

Step 2: Mark the end of a TLP episode and detect losses

If the sender receives a cumulative ACK for data beyond the TLP loss probe retransmission then, in the absence of reordering on the return path of ACKs, it should have received any ACKs for the original segment and TLP probe retransmission segment. At that time, if the TLPRxtOut flag is still true and thus indicates that the TLP probe retransmission remains unacknowledged, then the sender should presume that at least one of its data segments was lost, so it SHOULD invoke a congestion control response equivalent to fast recovery.

More precisely, on each ACK the sender executes the following:

```
If TLPRxtOut and SEG.ACK >= TLPHighRxt:
    TLPRxtOut = false
    EnterRecovery()
    ExitRecovery()
```

7. RACK and TLP discussions

7.1. Advantages

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

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

Example: LOST RETRANSMIT. Consider a window of three data packets (P1, P2, P3) that are sent; P1 and P2 are dropped. Suppose the transmission of each packet is at least RACK.reo_wnd (1 millisecond by default) after the transmission of the previous packet. When P3 is SACKed, RACK will mark P1 and P2 lost and they will be retransmitted as R1 and R2. Suppose R1 is lost again but R2 is SACKed; RACK will mark R1 lost for retransmission again. Again, neither the conventional three duplicate ACK threshold approach, nor [RFC6675], nor the Forward Acknowledgment [FACK] algorithm can detect such losses. And such a lost retransmission is very common when TCP is being rate-limited, particularly by token bucket policers with large bucket depth and low rate limit. Retransmissions are often lost repeatedly because standard congestion control requires multiple round trips to reduce the rate below the policed rate.

Example: SMALL DEGREE OF REORDERING. Consider a common reordering event: a window of packets are sent as (P1, P2, P3). P1 and P2 carry a full payload of MSS octets, but P3 has only a 1-octet payload.

Suppose the sender has detected reordering previously (e.g., by implementing the algorithm in [REORDER-DETECT]) and thus `RACK.reo_wnd` is `min_RTT/4`. Now P3 is reordered and delivered first, before P1 and P2. As long as P1 and P2 are delivered within `min_RTT/4`, RACK will not consider P1 and P2 lost. But if P1 and P2 are delivered outside the reordering window, then RACK will still falsely mark P1 and P2 lost. We discuss how to reduce false positives in the end of this section.

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

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

7.2. Disadvantages

RACK requires the sender to record the transmission time of each packet sent at a clock granularity of one millisecond or finer. TCP implementations that record this already for RTT estimation do not require any new per-packet state. But implementations that are not yet recording packet transmission times will need to add per-packet internal state (commonly either 4 or 8 octets per packet or TSO blob) to track transmission times. In contrast, the conventional [RFC6675] loss detection approach does not require any per-packet state beyond the SACK scoreboard.

7.3. Adjusting the reordering window

When the sender detects packet reordering, RACK uses a reordering window of `min_rtt / 4`. It uses the minimum RTT to accommodate reordering introduced by packets traversing slightly different paths (e.g., router-based parallelism schemes) or out-of-order deliveries in the lower link layer (e.g., wireless links using link-layer retransmission). RACK uses a quarter of minimum RTT because Linux TCP used the same factor in its implementation to delay Early Retransmit [RFC5827] to reduce spurious loss detections in the presence of reordering, and experience shows that this seems to work reasonably well. We have evaluated using the smoothed RTT (SRTT from

[RFC6298] RTT estimation) or the most recently measured RTT (RACK.RTT) using an experiment similar to that in the Performance Evaluation section. They do not make any significant difference in terms of total recovery latency.

One potential improvement would be 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.

7.4. Relationships with other loss recovery algorithms

The primary motivation of RACK is to ultimately provide a simple and general replacement for some of the standard loss recovery algorithms [RFC5681][RFC6675][RFC5827][RFC4653], as well as some nonstandard ones [FACK][THIN-STREAM]. While RACK can be a supplemental loss detection mechanism on top of these algorithms, this is not necessary, because RACK implicitly subsumes most of them.

[RFC5827][RFC4653][THIN-STREAM] dynamically adjusts the duplicate ACK threshold based on the current or previous flight sizes. RACK takes a different approach, by using only one ACK event and a reordering window. RACK can be seen as an extended Early Retransmit [RFC5827] without a FlightSize limit but with an additional reordering window. [FACK] considers an original packet to be lost when its sequence range is sufficiently far below the highest SACKed sequence. In some sense RACK can be seen as a generalized form of FACK that operates in time space instead of sequence space, enabling it to better handle reordering, application-limited traffic, and lost retransmissions.

Nevertheless RACK is still an experimental algorithm. Since the oldest loss detection algorithm, the 3 duplicate ACK threshold [RFC5681], has been standardized and widely deployed, 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 RTO timer calculation nor detects spurious timeouts.

Furthermore, RACK naturally works well with Tail Loss Probe [TLP] because a tail loss probe solicits either an ACK or SACK, which can be used by RACK to detect more losses. RACK can be used to relax TLP's requirement for using FACK and retransmitting the the highest-sequenced packet, because RACK is agnostic to packet sequence numbers, and uses transmission time instead. Thus TLP could be modified to retransmit the first unacknowledged packet, which could improve application latency.

7.5. Interaction with congestion control

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

RACK is applicable for both fast recovery and recovery after a retransmission timeout (RTO) in [RFC5681]. RACK applies equally to fast recovery and RTO recovery because RACK is purely based on the transmission time order of packets. When a packet retransmitted by RTO is acknowledged, RACK will mark any unacked packet sent sufficiently prior to the RTO as lost, because at least one RTT has elapsed since these packets were sent.

The following simple example compares how RACK and non-RACK loss detection interacts with congestion control: suppose a TCP sender has a congestion window (cwnd) of 20 packets on a SACK-enabled connection. It sends 10 data packets and all of them are lost.

Without RACK, the sender would time out, reset cwnd to 1, and retransmit first packet. It would take another four round trips ($1 + 2 + 4 + 3 = 10$) to retransmit all the 10 lost packets using slow start. The recovery latency would be $RTO + 4*RTT$, with an ending cwnd of 4 packets due to congestion window validation.

With RACK, a sender would send the TLP after $2*RTT$ and get a DUPACK. If the sender implements Proportional Rate Reduction [RFC6937] it would slow start to retransmit the remaining 9 lost packets since the number of packets in flight (0) is lower than the slow start threshold (10). The slow start would again take another four round trips ($1 + 2 + 4 + 3 = 10$). The recovery latency would be $2*RTT + 4*RTT$, with an ending cwnd set to the slow start threshold of 10 packets.

In both cases, the sender after the recovery would be in congestion avoidance. The difference in recovery latency ($RTO + 4*RTT$ vs $6*RTT$) can be significant if the RTT is much smaller than the minimum RTO (1 second in RFC6298) or if the RTT is large. The former case is common in local area networks, data-center networks, or content distribution networks with deep deployments. The latter case is more common in developing regions with highly congested and/or high-latency networks. The ending congestion window after recovery also impacts subsequent data transfer.

7.6. TLP recovery detection with delayed ACKs

Delayed ACKs complicate the detection of repairs done by TLP, since with a delayed ACK the sender receives one fewer ACK than would normally be expected. To mitigate this complication, before sending a TLP loss probe retransmission, the sender should attempt to wait long enough that the receiver has sent any delayed ACKs that it is withholding. The sender algorithm described above features such a delay, in the form of `WCDelAckT`. Furthermore, if the receiver supports duplicate selective acknowledgments (D-SACKs) [RFC2883] then in the case of a delayed ACK the sender's TLP recovery detection algorithm (see above) can use the D-SACK information to infer that the original and TLP retransmission both arrived at the receiver.

If there is ACK loss or a delayed ACK without a D-SACK, then this algorithm is conservative, because the sender will reduce `cwnd` when in fact there was no packet loss. In practice this is acceptable, and potentially even desirable: if there is reverse path congestion then reducing `cwnd` can be prudent.

7.7. RACK for other transport protocols

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

8. Experiments and Performance Evaluations

RACK and TLP have been deployed at Google, for both connections to users in the Internet and internally. We conducted a performance evaluation experiment for RACK and TLP on a small set of Google Web servers in Western Europe that serve mostly European and some African countries. The experiment lasted three days in March 2017. The servers were divided evenly into four groups of roughly 5.3 million flows each:

Group 1 (control): RACK off, TLP off, RFC 3517 on

Group 2: RACK on, TLP off, RFC 3517 on

Group 3: RACK on, TLP on, RFC 3517 on

Group 4: RACK on, TLP on, RFC 3517 off

All groups used Linux with CUBIC congestion control, an initial congestion window of 10 packets, and the fq/pacing qdisc. In terms of specific recovery features, all groups enabled RFC5682 (F-RTO) but disabled FACK because it is not an IETF RFC. FACK was excluded because the goal of this setup is to compare RACK and TLP to RFC-based loss recoveries. Since TLP depends on either FACK or RACK, we could not run another group that enables TLP only (with both RACK and FACK disabled). Group 4 is to test whether RACK plus TLP can completely replace the DupThresh-based [RFC3517].

The servers sit behind a load balancer that distributes the connections evenly across the four groups.

Each group handles a similar number of connections and sends and receives similar amounts of data. We compare total time spent in loss recovery across groups. The recovery time is measured from when the recovery and retransmission starts, until the remote host has acknowledged the highest sequence (SND.NXT) at the time the recovery started. Therefore the recovery includes both fast recoveries and timeout recoveries.

Our data shows that Group 2 recovery latency is only 0.3% lower than the Group 1 recovery latency. But Group 3 recovery latency is 25% lower than Group 1 due to a 40% reduction in RTO-triggered recoveries! Therefore it is important to implement both TLP and RACK for performance. Group 4's total recovery latency is 0.02% lower than Group 3's, indicating that RACK plus TLP can successfully replace RFC3517 as a standalone recovery mechanism.

We want to emphasize that the current experiment is limited in terms of network coverage. The connectivity in Western Europe is fairly good, therefore loss recovery is not a major performance bottleneck. We plan to expand our experiments to regions with worse connectivity, in particular on networks with strong traffic policing.

9. Security Considerations

RACK does not change the risk profile for TCP.

An interesting scenario is ACK-splitting attacks [SCWA99]: for an MSS-size packet sent, the receiver or the attacker might send MSS ACKs that SACK or acknowledge one additional byte per ACK. This

would not fool RACK. RACK.xmit_ts would not advance because all the sequences of the packet are transmitted at the same time (carry the same transmission timestamp). In other words, SACKing only one byte of a packet or SACKing the packet in entirety have the same effect on RACK.

10. IANA Considerations

This document makes no request of IANA.

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

11. Acknowledgments

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

12. References

12.1. 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., Dukkupati, 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.

12.2. Informative References

- [FACK] Mathis, M. and M. Jamshid, "Forward acknowledgement: refining TCP congestion control", ACM SIGCOMM Computer Communication Review, Volume 26, Issue 4, Oct. 1996. , 1996.
- [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

Nandita Dukkkipati
Google, Inc
1600 Amphitheater Parkway
Mountain View, California 94043
USA

Email: nanditad@google.com

Internet Engineering Task Force
Internet-Draft
Obsoletes: 793, 879, 6093, 6429, 6528,
6691 (if approved)
Updates: 5961, 1122 (if approved)
Intended status: Standards Track
Expires: January 18, 2018

W. Eddy, Ed.
MTI Systems
July 17, 2017

Transmission Control Protocol Specification
draft-ietf-tcpm-rfc793bis-06

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, 6093, 6429, 6528, and 6691. 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. (TODO: double-check this list for all actual RFCs when finished)

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

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

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

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

This Internet-Draft will expire on January 18, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

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

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1. Purpose and Scope	3
2. Introduction	4
3. Functional Specification	5
3.1. Header Format	5
3.2. Terminology	10
3.3. Sequence Numbers	15
3.4. Establishing a connection	21
3.4.1. Remote Address Validation	28
3.5. Closing a Connection	28
3.5.1. Half-Closed Connections	31
3.6. Precedence and Security	31
3.7. Segmentation	32

3.7.1.	Maximum Segment Size Option	33
3.7.2.	Path MTU Discovery	35
3.7.3.	Interfaces with Variable MTU Values	35
3.7.4.	Nagle Algorithm	36
3.7.5.	IPv6 Jumbograms	36
3.8.	Data Communication	36
3.8.1.	Retransmission Timeout	37
3.8.2.	TCP Congestion Control	37
3.8.3.	TCP Connection Failures	38
3.8.4.	TCP Keep-Alives	39
3.8.5.	The Communication of Urgent Information	39
3.8.6.	Managing the Window	40
3.9.	Interfaces	44
3.9.1.	User/TCP Interface	45
3.9.2.	TCP/Lower-Level Interface	53
3.10.	Event Processing	55
3.11.	Glossary	80
4.	Changes from RFC 793	86
5.	IANA Considerations	90
6.	Security and Privacy Considerations	90
7.	Acknowledgements	90
8.	References	91
8.1.	Normative References	91
8.2.	Informative References	92
Appendix A.	Other Implementation Notes	93
Appendix B.	TCP Requirement Summary	94
Author's Address	97

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 [27]. 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 describes 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 examples and other discussion in RFC 793. 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" [27] 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. [18]) 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 [26], but these are not strictly required or discussed in this document.

TEMPORARY EDITOR'S NOTE: This is an early revision in the process of updating RFC 793. Many planned changes are not yet incorporated.

Please do not use this revision as a basis for any work or reference.

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

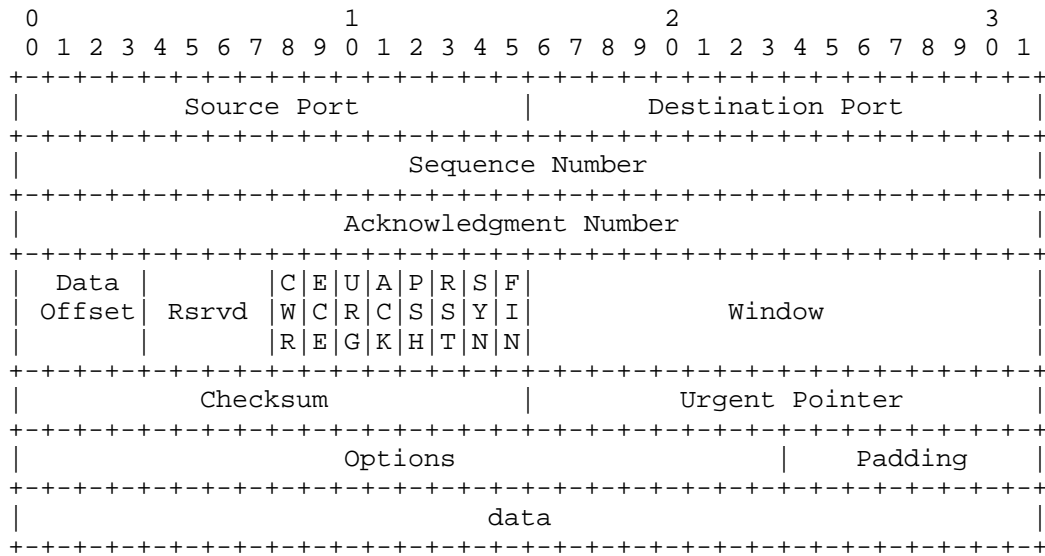
TEMPORARY EDITOR'S NOTE: the current revision of this document does not yet collect all of the changes that will be in the final version. The set of content changes planned for future revisions is kept in Section 4.

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] [5]. 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. (Editorial TODO - this last sentence makes sense in 793 context, but may be a candidate to remove here? ... additionally, Section 2 of 793 is not included here, but some parts may be useful, to quickly define basic concepts of ports, bytestream service, etc. at high-level before delving into protocol details?)

TCP Header Format



TCP Header Format

Note that one tick mark represents one bit position.

Figure 1

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.

Rsvrd - Reserved: 4 bits

Reserved for future use. Must be zero in generated segments and must be ignored in received segments. TODO -- no RFC reference for this sentence ... do we want this change or should we keep the prior 793 description which is only "Must be zero." ... need to discuss on TCPM list

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

- CWR: Congestion Window Reduced (see [9])
- ECE: ECN-Echo (see [9])
- URG: Urgent Pointer field significant
- ACK: Acknowledgment field significant
- PSH: Push Function
- RST: Reset the connection
- SYN: Synchronize sequence numbers
- FIN: No more data from sender

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.

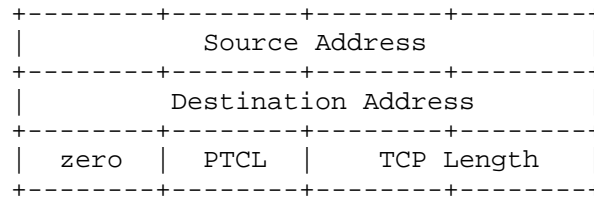
The window size MUST be treated as an unsigned number, or else large window sizes will appear like negative windows and TCP will now work. It is RECOMMENDED that implementations will reserve 32-bit fields for the send and receive window sizes in the connection record and do all window computations with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

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, 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.



The TCP Length is 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 2460 [5], 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 and the receiver MUST check it.

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 [29], 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 [25].

A given TCP implementation can support any currently defined options, but the following options MUST be supported (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. A TCP MUST 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.

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

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 security and precedence of the connection, 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

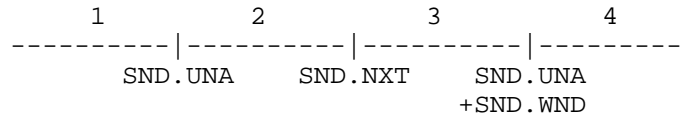
SND.UNA - send unacknowledged
 SND.NXT - send next
 SND.WND - send window
 SND.UP - send urgent pointer
 SND.WL1 - segment sequence number used for last window update
 SND.WL2 - segment acknowledgment number used for last window update
 ISS - initial send sequence number

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



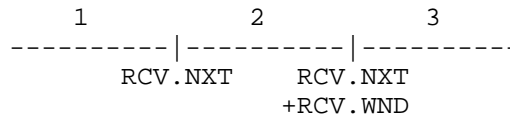
- 1 - old sequence numbers which have been acknowledged
- 2 - sequence numbers of unacknowledged data
- 3 - sequence numbers allowed for new data transmission
- 4 - future sequence numbers which are not yet allowed

Send Sequence Space

Figure 2

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

Receive Sequence Space



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

Receive Sequence Space

Figure 3

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
- SEG.PRC - segment precedence value

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).

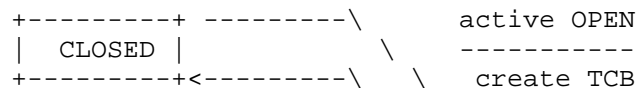
TIME-WAIT - represents waiting for enough time to pass to be sure the remote TCP received the acknowledgment of its connection termination request.

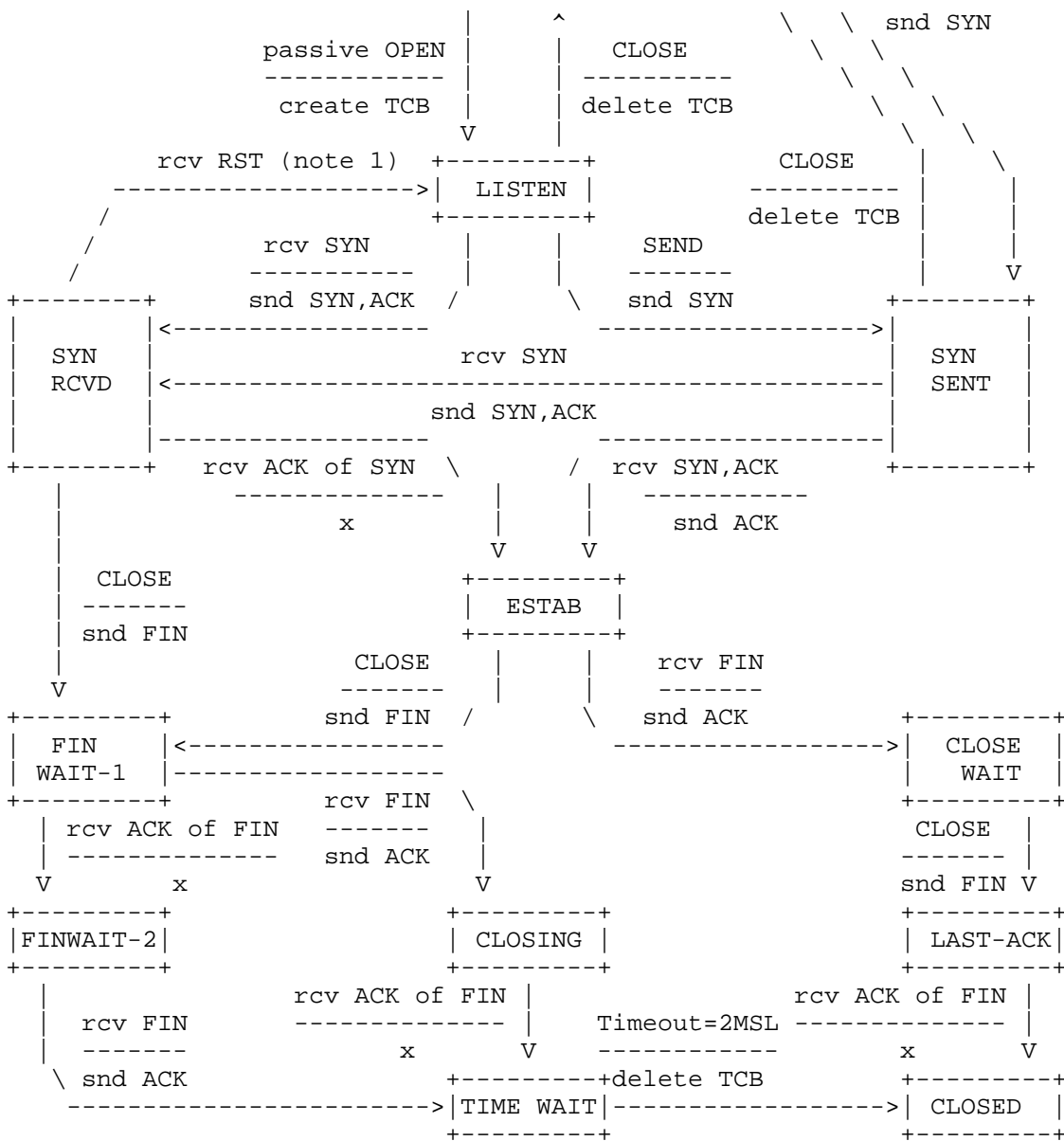
CLOSED - represents no connection state at all.

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.





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.

TCP Connection State Diagram

Figure 4

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:

$$\text{SND.UNA} < \text{SEG.ACK} \leq \text{SND.NXT}$$

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

$$\text{RCV.NXT} \leq \text{SEG.SEQ} < \text{RCV.NXT} + \text{RCV.WND}$$

or

$$\text{RCV.NXT} \leq \text{SEG.SEQ} + \text{SEG.LEN} - 1 < \text{RCV.NXT} + \text{RCV.WND}$$

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 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, and SHOULD generate its Initial Sequence Numbers with the expression:

$$\text{ISN} = M + F(\text{localip}, \text{localport}, \text{remoteip}, \text{remoteport}, \text{secretkey})$$

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"). F() MUST NOT be computable from the outside, 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 hash 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 [23].

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 it's 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 [3].

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 S_1 , 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 $S_1 = ISN(t)$ -- 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 S_1 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

Basic 3-Way Handshake for Connection Synchronization

Figure 5

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

Simultaneous Connection Synchronization

Figure 6

A TCP MUST support simultaneous open attempts.

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.

The principle 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

Recovery from Old Duplicate SYN

Figure 7

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>	-->

Half-Open Connection Discovery

Figure 8

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!!)

Active Side Causes Half-Open Connection Discovery

Figure 9

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

Old Duplicate SYN Initiates a Reset on two Passive Sockets

Figure 10

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 our SYN has not been acknowledged and the precedence level of the incoming segment is higher than the precedence level requested then either raise the local precedence level (if allowed by the user and the system) or send a reset; or if the precedence level of the incoming segment is lower than the precedence level requested then continue as if the precedence matched exactly (if the remote TCP cannot raise the precedence level to match ours this will be detected in the next segment it sends, and the connection will be terminated then). If our SYN has been acknowledged (perhaps in this incoming segment) the precedence level of the incoming segment must match the local precedence level exactly, if it does not a reset must be 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, or precedence which does not exactly match the level, and compartment, and precedence 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.

3.4.1. Remote Address Validation

TODO - figure out if this section would fit better elsewhere, for instance in the more detailed description of the OPEN call later on

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).

An incoming SYN with an invalid source address must be ignored either by TCP or by the IP layer (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.

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. TIME-WAIT	<-- <SEQ=300><ACK=101><CTL=FIN,ACK>	(Close) <-- LAST-ACK
5. TIME-WAIT	--> <SEQ=101><ACK=301><CTL=ACK>	--> CLOSED
6. (2 MSL) CLOSED		

Normal Close Sequence

Figure 11

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 (2 MSL) CLOSED		TIME-WAIT (2 MSL) CLOSED

Simultaneous Close Sequence

Figure 12

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 a TCP connection is closed by the remote site, the local application MUST be informed whether it closed normally or was aborted.

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. 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.

When a connection is closed actively, it MUST linger in TIME-WAIT state for a time $2 \times \text{MSL}$ (Maximum Segment Lifetime). However, it MAY accept a new SYN from the remote TCP to reopen the connection directly from TIME-WAIT state, 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.

3.6. Precedence and Security

TODO - talk to TCPM about what to do about precedence and security compartment throughout the document ... security compartment material for IPv4 may be fine nearly as-is, but precedence was a subset of what DSCP includes and it's not clear that running code actually does what 793 says about precedence anyways, especially since now as a DSCP it doesn't make sense to do greater-than comparisons on, nor to reset connections if it changes.

The intent is that connection be allowed only between ports operating with exactly the same security and compartment values and at the higher of the precedence level requested by the two ports.

The precedence and security parameters used in TCP are exactly those defined in the Internet Protocol (IP) [1]. Throughout this TCP

specification the term "security/compartment" is intended to indicate the security parameters used in IP including security, compartment, user group, and handling restriction.

A connection attempt with mismatched security/compartment values or a lower precedence value must be rejected by sending a reset. Rejecting a connection due to too low a precedence only occurs after an acknowledgment of the SYN has been received.

Note that TCP modules which operate only at the default value of precedence will still have to check the precedence of incoming segments and possibly raise the precedence level they use on the connection.

The security parameters may be used even in a non-secure environment (the values would indicate unclassified data), thus hosts in non-secure environments must be prepared to receive the security parameters, though they need not send them.

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 [16], 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 structures, 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 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.

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, and MAY send it always.

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.

The maximum size of a segment that TCP really sends, the "effective send MSS," MUST be the smaller 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 =

$$\min(\text{SendMSS}+20, \text{MMS_S}) - \text{TCPHdrsize} - \text{IPOptionsize}$$

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 [24] discusses this in greater detail.

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

$$\text{MMS_R} - 20$$

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 [8]. PLPMTUD [15] 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) [19]. 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.

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.

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).

TODO - see if SEND description later should be updated to reflect this

A TCP SHOULD implement the Nagle Algorithm to coalesce short segments. However, there MUST be a way for an application to disable the Nagle algorithm on an individual connection. In all cases, sending data is also subject to the limitation imposed by the Slow Start algorithm [18].

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 [7] 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 [10], including Karn's algorithm for taking RTT samples.

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).

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.

Explicit Congestion Notification (ECN) was defined in RFC 3168 and is an IETF Standards Track enhancement that has many benefits [28].

A TCP SHOULD implement ECN as described in RFC 3168.

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:

(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 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.

(d) TCP SHOULD inform the application of the delivery problem (unless such information has been disabled by the application; see RFC1122 Section 4.2.4.1 - TODO update to error reporting description in this document), when R1 is reached and before R2. 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. The value of R2 SHOULD correspond to at least 100 seconds.

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, 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, and they MUST default to off.

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

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.

An implementation SHOULD send a keep-alive segment with no data; however, it MAY be configurable to send a keep-alive segment containing one garbage octet, 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. However, TCP implementations MUST still include support for the urgent mechanism. Details can be found in RFC 6093 [21].

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. [14]

A TCP MUST inform the application layer asynchronously whenever it receives an Urgent pointer and there was previously no pending urgent data, or whenever the Urgent pointer advances in the data stream. There MUST 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 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. 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.

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

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.

A TCP MAY keep its offered receive window closed indefinitely. 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. 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 [22].

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).

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.

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

The sender's SWS avoidance algorithm is more difficult than the receiver'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 $\text{Max}(\text{SND.WND})$, 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:

$$U = \text{SND.UNA} + \text{SND.WND} - \text{SND.NXT}$$

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:

$$\min(D,U) \geq \text{Eff.snd.MSS};$$

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

$$[\text{SND.NXT} = \text{SND.UNA} \text{ and}] \text{ PUSHED} \text{ and } D \leq U$$

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

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

[SND.NXT = SND.UNA and]

$\min(D.U) \geq F_s * \text{Max}(\text{SND.WND});$

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

Here F_s 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

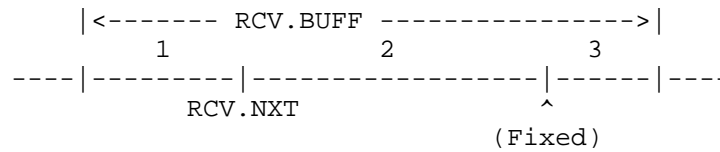
A TCP MUST include a SWS avoidance algorithm in the receiver.

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 $\text{RCV.NXT} + \text{RCV.WND}$ 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, $\text{RCV.WND} = \text{RCV.BUFF}$ and $\text{RCV.USER} = 0$.

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 $\text{RCV.NXT} + \text{RCV.WND}$ constant as RCV.NXT increases. Thus, the total buffer space RCV.BUFF is generally divided into three parts:



- 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:

$$\text{RCV.BUFF} - \text{RCV.USER} - \text{RCV.WND} \geq \min(\text{Fr} * \text{RCV.BUFF}, \text{Eff.snd.MSS})$$

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, but an ACK should not be excessively delayed; in particular, the delay MUST be less than 0.5 seconds, and in a stream of full-sized segments there SHOULD be an ACK for at least every second segment. 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 (e.g., SVCs, UUOs, EMTs).

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] [, precedence] [, security/compartments] [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.

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.

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 precedence or security/compartments. The absence of precedence or security/compartments specification in the OPEN call indicates the default values must be used.

TCP will accept incoming requests as matching only if the security/compartments information is exactly the same and only

if the precedence is equal to or higher than the precedence requested in the OPEN call.

The precedence for the connection is the higher of the values requested in the OPEN call and received from the incoming request, and fixed at that value for the life of the connection. Implementers may want to give the user control of this precedence negotiation. For example, the user might be allowed to specify that the precedence must be exactly matched, or that any attempt to raise the precedence be confirmed by the user.

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. 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 MUST be used for opening the connection. If the parameter is unspecified, the TCP will choose an appropriate local IP address (see RFC 1122 section 3.3.4.2).

TODO - the previous and next paragraphs are mildly in conflict. Previous paragraph says that the TCP chooses an address, but next paragraph says that it asks IP to choose ... need to make this consistent

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. 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.

Send

Format: SEND (local connection name, buffer address, byte count, PUSH flag, 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. If the calling process is not authorized to use this connection, an error is returned.

If the PUSH flag is set, the data must be transmitted promptly to the receiver, and the PUSH bit will be set in the last TCP segment created from the buffer. If the PUSH flag is not set, the data may be combined with data from subsequent SENDS for transmission efficiency.

New applications SHOULD NOT set the URGENT flag [21] due to implementation differences and middlebox issues.

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

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.

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,
- precedence,
- security/compartments,
- 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.

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. 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. TCP SHOULD pass the current Differentiated Services field value without change to the IP layer, when it sends segments on the connection.

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.

TCP-to-User Messages

It is assumed that the operating system environment provides a means for the TCP to asynchronously signal the user program. When the TCP does signal a user program, certain information is passed to the user. Often in the specification the information will be an error message. In other cases there will be

information relating to the completion of processing a SEND or RECEIVE or other user call.

The following information is provided:

Local Connection Name	Always
Response String	Always
Buffer Address	Send & Receive
Byte count (counts bytes received)	Receive
Push flag	Receive
Urgent flag	Receive

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 [5].

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:

Type of Service = Precedence: given by user, Delay: normal, Throughput: normal, Reliability: normal; or binary XXX00000, where XXX are the three bits determining precedence, e.g. 000 means routine precedence. TODO - this is pretty much wrong with regard to DiffServ, I think we should just say that the user can specify diffserv field (superset of DSCP) and leave it at that, but will check with TCPM

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

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.

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.

A TCP MAY support the Time Stamp and Record Route 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, and this MUST take precedence over a source route received in a datagram.

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. If a different source route arrives in a later segment, the later definition SHOULD override the earlier one.

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. 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.

[17] 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. See [11] 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, and it SHOULD make the information available to the application.

Hard Errors

For ICMP these include Destination Unreachable -- codes 2-4">
These are hard error conditions, so TCP SHOULD abort the connection. [17] 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 [17] section 4 describes widespread implementation behavior that treats soft errors as hard errors during connection establishment.

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, precedence, security/compartments, 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 precedence requested are allowed for this user, if not return "error: precedence not allowed" or "error: security/compartments 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 $\text{SND.UP} \leftarrow \text{SND.NXT}$ 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>

If the SEG.PRC is greater than the TCB.PRC then if allowed by the user and the system set TCB.PRC<-SEG.PRC, if not allowed send a reset and return.

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

If the SEG.PRC is less than the TCB.PRC then continue.

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. (TODO: in processing Errata ID 3300, it was noted that some stacks in the wild that do not send data on the SYN are just checking that `SEG.ACK == SND.NXT ...` think about whether anything should be said about that here)

second check the RST bit

If the RST bit is set

A potential blind reset attack is described in RFC 5961 [20], 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 and precedence

If the security/compartiment in the segment does not exactly match the security/compartiment 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 there is an ACK

The precedence in the segment must match the precedence in the TCB, if not, send a reset

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

If there is no ACK

If the precedence in the segment is higher than the precedence in the TCB then if allowed by the user and the system raise the precedence in the TCB to that in the

segment, if not allowed to raise the prec then send a reset.

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

If the precedence in the segment is lower than the precedence in the TCB continue.

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 if the segment did not contain a RST.

If the SYN bit is on and the security/compartments and precedence are acceptable then, RCV.NXT is set to SEG.SEQ+1, IRS is set to SEG.SEQ. 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.

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.

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

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

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

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

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

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

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

In general, the processing of received segments MUST be implemented to aggregate ACK segments whenever possible. For example, if the TCP is processing a series of queued segments, it MUST process them all before sending any ACK segments. (TODO - see if there's a better place for this paragraph - taken from RFC1122)

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 processing 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 and precedence

SYN-RECEIVED

If the security/compartments and precedence in the segment do not exactly match the security/compartments and precedence 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/compartments and precedence in the segment do not exactly match the security/compartments and precedence 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 or precedence 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 an error where the connection should be reset, or the result of an attack attempt, as described in RFC 5961 [20]. 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. 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 $((\text{SND.UNA} - \text{MAX.SND.WND}) \leq \text{SEG.ACK} \leq \text{SND.NXT})$. 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 $\text{SND.UNA} < \text{SEG.ACK} \leq \text{SND.NXT}$ 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 $\text{SND.UNA} < \text{SEG.ACK} \leq \text{SND.NXT}$ then, set $\text{SND.UNA} \leftarrow \text{SEG.ACK}$. 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 ($\text{SEG.ACK} \leq \text{SND.UNA}$), it can be ignored. If the ACK acks something not yet sent ($\text{SEG.ACK} > \text{SND.NXT}$) then send an ACK, drop the segment, and return.

If $\text{SND.UNA} \leq \text{SEG.ACK} \leq \text{SND.NXT}$, the send window should be updated. If ($\text{SND.WL1} < \text{SEG.SEQ}$ or ($\text{SND.WL1} = \text{SEG.SEQ}$ and $\text{SND.WL2} \leq \text{SEG.ACK}$)), set $\text{SND.WND} \leftarrow \text{SEG.WND}$, set $\text{SND.WL1} \leftarrow \text{SEG.SEQ}$, and set $\text{SND.WL2} \leftarrow \text{SEG.ACK}$.

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, $RCV.UP \leftarrow \max(RCV.UP, SEG.UP)$, 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.

Please note the window management suggestions in section 3.7.

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

1822 BBN Report 1822, "The Specification of the Interconnection of a Host and an IMP". The specification of interface between a host and the ARPANET.

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.

ARPANET message

The unit of transmission between a host and an IMP in the ARPANET. The maximum size is about 1012 octets (8096 bits).

ARPANET packet

A unit of transmission used internally in the ARPANET between IMPs. The maximum size is about 126 octets (1008 bits).

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.

FTP

A file transfer protocol.

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.
- IMP**
The Interface Message Processor, the packet switch of the ARPANET.
- 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.
- 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.
- leader**
Control information at the beginning of a message or block of data. In particular, in the ARPANET, the control information on an ARPANET message at the host-IMP interface.
- 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.

local packet

The unit of transmission within a local network.

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. The options are used primarily in testing situations; for example, to carry timestamps. Both the Internet Protocol and TCP provide for options fields.

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.

RTP
Real Time Protocol: A host-to-host protocol for communication of time critical information.

SEG.ACK
segment acknowledgment

SEG.LEN
segment length

SEG.PRC
segment precedence value

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 transferred 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

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.

TCB.PRC

The precedence of the connection.

TCP

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

TOS

Type of Service, an IPv4 field, that currently carries the Differentiated Services field [6] 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 the informational text with background and rationale has not 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, 3602). TODO: 3305

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 management 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
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.
There is an errata 3305 currently reported that need to be verified, held, or rejected by the ADs; it is addressing the same issue as draft-gont-tcpm-tcp-seq-validation and was not attempted to be applied to this document.

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 formatting 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 implement 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.

TODO list of other planned changes (these can be added to or made more specific, as the document proceeds):

1. mention 6161 (reducing TIME-WAIT)
2. clarify data on SYN from Michael Welzl

Some other suggested changes that will not be incorporated in this 793 update unless TCPM consensus changes with regard to scope are:

1. look at Tony Sabatini suggestion for describing DO field
2. clearly specify treatment of reserved bits (see TCPM thread on EDO draft April 25, 2014) -- TODO - an attempt at this is actually in -06, but needs to be confirmed by TCPM explicitly since there is no RFC reference
3. per discussion with Joe Touch (TAPS list, 6/20/2015), the description of the API could be revisited

5. IANA Considerations

This memo includes no request to IANA. Existing IANA registries for TCP parameters are sufficient.

TODO: check whether entries pointing to 793 and other documents obsoleted by this one should be updated to point to this one instead.

6. Security and Privacy Considerations

TODO

See RFC 6093 [21] for discussion of security considerations related to the urgent pointer field.

Editor's Note: Scott Brim mentioned that this should include a PERPASS/privacy review.

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:

Michael Scharf
Yoshifumi Nishida
Pasi Sarolahti

During early discussion of this work on the TCPM mailing list, and at the IETF 88 meeting in Vancouver, helpful comments, critiques, and reviews were received from (listed alphabetically): David Borman, Yuchung Cheng, Martin Duke, Kevin Lahey, Kevin Mason, Matt Mathis, Hagen Paul Pfeifer, Anthony Sabatini, Joe Touch, Reji Varghese, Lloyd Wood, and Alex Zimmermann. Joe Touch provided 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, <<http://www.rfc-editor.org/info/rfc791>>.
- [2] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, DOI 10.17487/RFC1191, November 1990, <<http://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, <<http://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, <<http://www.rfc-editor.org/info/rfc2119>>.
- [5] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, DOI 10.17487/RFC2460, December 1998, <<http://www.rfc-editor.org/info/rfc2460>>.
- [6] 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, <<http://www.rfc-editor.org/info/rfc2474>>.
- [7] Borman, D., Deering, S., and R. Hinden, "IPv6 Jumbograms", RFC 2675, DOI 10.17487/RFC2675, August 1999, <<http://www.rfc-editor.org/info/rfc2675>>.
- [8] Lahey, K., "TCP Problems with Path MTU Discovery", RFC 2923, DOI 10.17487/RFC2923, September 2000, <<http://www.rfc-editor.org/info/rfc2923>>.
- [9] 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>>.
- [10] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, DOI 10.17487/RFC6298, June 2011, <<http://www.rfc-editor.org/info/rfc6298>>.

- [11] Gont, F., "Deprecation of ICMP Source Quench Messages", RFC 6633, DOI 10.17487/RFC6633, May 2012, <<http://www.rfc-editor.org/info/rfc6633>>.

8.2. Informative References

- [12] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <<http://www.rfc-editor.org/info/rfc793>>.
- [13] Nagle, J., "Congestion Control in IP/TCP Internetworks", RFC 896, DOI 10.17487/RFC0896, January 1984, <<http://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, <<http://www.rfc-editor.org/info/rfc1122>>.
- [15] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, DOI 10.17487/RFC4821, March 2007, <<http://www.rfc-editor.org/info/rfc4821>>.
- [16] 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, <<http://www.rfc-editor.org/info/rfc5044>>.
- [17] Gont, F., "TCP's Reaction to Soft Errors", RFC 5461, DOI 10.17487/RFC5461, February 2009, <<http://www.rfc-editor.org/info/rfc5461>>.
- [18] 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>>.
- [19] Sandlund, K., Pelletier, G., and L-E. Jonsson, "The RObust Header Compression (ROHC) Framework", RFC 5795, DOI 10.17487/RFC5795, March 2010, <<http://www.rfc-editor.org/info/rfc5795>>.
- [20] Ramaiah, A., Stewart, R., and M. Dalal, "Improving TCP's Robustness to Blind In-Window Attacks", RFC 5961, DOI 10.17487/RFC5961, August 2010, <<http://www.rfc-editor.org/info/rfc5961>>.

- [21] Gont, F. and A. Yourtchenko, "On the Implementation of the TCP Urgent Mechanism", RFC 6093, DOI 10.17487/RFC6093, January 2011, <<http://www.rfc-editor.org/info/rfc6093>>.
- [22] Bashyam, M., Jethanandani, M., and A. Ramaiah, "TCP Sender Clarification for Persist Condition", RFC 6429, DOI 10.17487/RFC6429, December 2011, <<http://www.rfc-editor.org/info/rfc6429>>.
- [23] Gont, F. and S. Bellovin, "Defending against Sequence Number Attacks", RFC 6528, DOI 10.17487/RFC6528, February 2012, <<http://www.rfc-editor.org/info/rfc6528>>.
- [24] Borman, D., "TCP Options and Maximum Segment Size (MSS)", RFC 6691, DOI 10.17487/RFC6691, July 2012, <<http://www.rfc-editor.org/info/rfc6691>>.
- [25] Touch, J., "Shared Use of Experimental TCP Options", RFC 6994, DOI 10.17487/RFC6994, August 2013, <<http://www.rfc-editor.org/info/rfc6994>>.
- [26] 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>>.
- [27] 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, <<http://www.rfc-editor.org/info/rfc7414>>.
- [28] Fairhurst, G. and M. Welzl, "The Benefits of Using Explicit Congestion Notification (ECN)", RFC 8087, DOI 10.17487/RFC8087, March 2017, <<http://www.rfc-editor.org/info/rfc8087>>.
- [29] IANA, "Transmission Control Protocol (TCP) Parameters, <https://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml>", 2017.

Appendix A. Other Implementation Notes

TODO - mention draft-gont-tcpm-tcp-seccomp-prec - per IETF 99 TPCM discussion

TODO - mention draft-gont-tcpm-tcp-seq-validation - per IETF 99 TPCM discussion

Allow window stay zero indefinitely	4.2.2.17	x			
Sender timeout OK conn with zero wind	4.2.2.17				x
Urgent Data					
Pointer indicates first non-urgent octet	4.2.2.4	x			
Arbitrary length urgent data sequence	4.2.2.4	x			
Inform ALP asynchronously of urgent data	4.2.2.4	x			1
ALP can learn if/how much urgent data Q'd	4.2.2.4	x			1
TCP Options					
Receive TCP option in any segment	4.2.2.5	x			
Ignore unsupported options	4.2.2.5	x			
Cope with illegal option length	4.2.2.5	x			
Implement sending & receiving MSS option	4.2.2.6	x			
IPv4 Send MSS option unless 536	4.2.2.6		x		
IPv6 Send MSS option unless 1220	N/A		x		
Send MSS option always	4.2.2.6			x	
IPv4 Send-MSS default is 536	4.2.2.6	x			
IPv6 Send-MSS default is 1220	N/A		x		
Calculate effective send seg size	4.2.2.6	x			
MSS accounts for varying MTU	N/A			x	
TCP Checksums					
Sender compute checksum	4.2.2.7	x			
Receiver check checksum	4.2.2.7	x			
ISN Selection					
Include a clock-driven ISN generator component	4.2.2.9	x			
Secure ISN generator with a PRF component	N/A			x	
Opening Connections					
Support simultaneous open attempts	4.2.2.10	x			
SYN-RECEIVED remembers last state	4.2.2.11	x			
Passive Open call interfere with others	4.2.2.18				x
Function: simultan. LISTENS for same port	4.2.2.18	x			
Ask IP for src address for SYN if necc.	4.2.3.7	x			
Otherwise, use local addr of conn.	4.2.3.7	x			
OPEN to broadcast/multicast IP Address	4.2.3.14				x
Silently discard seg to bcast/mcast addr	4.2.3.14	x			
Closing Connections					
RST can contain data	4.2.2.12		x		
Inform application of aborted conn	4.2.2.13	x			
Half-duplex close connections	4.2.2.13			x	
Send RST to indicate data lost	4.2.2.13		x		
In TIME-WAIT state for 2MSL seconds	4.2.2.13	x			
Accept SYN from TIME-WAIT state	4.2.2.13			x	

Retransmissions						
Jacobson Slow Start algorithm	4.2.2.15	x				
Jacobson Congestion-Avoidance algorithm	4.2.2.15	x				
Retransmit with same IP ident	4.2.2.15			x		
Karn's algorithm	4.2.3.1	x				
Jacobson's RTO estimation alg.	4.2.3.1	x				
Exponential backoff	4.2.3.1	x				
SYN RTO calc same as data	4.2.3.1		x			
Recommended initial values and bounds	4.2.3.1		x			
Generating ACK's:						
Queue out-of-order segments	4.2.2.20		x			
Process all Q'd before send ACK	4.2.2.20	x				
Send ACK for out-of-order segment	4.2.2.21			x		
Delayed ACK's	4.2.3.2		x			
Delay < 0.5 seconds	4.2.3.2	x				
Every 2nd full-sized segment ACK'd	4.2.3.2	x				
Receiver SWS-Avoidance Algorithm	4.2.3.3	x				
Sending data						
Configurable TTL	4.2.2.19	x				
Sender SWS-Avoidance Algorithm	4.2.3.4	x				
Nagle algorithm	4.2.3.4		x			
Application can disable Nagle algorithm	4.2.3.4	x				
Connection Failures:						
Negative advice to IP on R1 retxs	4.2.3.5	x				
Close connection on R2 retxs	4.2.3.5	x				
ALP can set R2	4.2.3.5	x				1
Inform ALP of R1<=retxs<R2	4.2.3.5		x			1
Recommended values for R1, R2	4.2.3.5		x			
Same mechanism for SYN's	4.2.3.5	x				
R2 at least 3 minutes for SYN	4.2.3.5	x				
Send Keep-alive Packets:						
	4.2.3.6			x		
- Application can request	4.2.3.6	x				
- Default is "off"	4.2.3.6	x				
- Only send if idle for interval	4.2.3.6	x				
- Interval configurable	4.2.3.6	x				
- Default at least 2 hrs.	4.2.3.6	x				
- Tolerant of lost ACK's	4.2.3.6	x				
IP Options						
Ignore options TCP doesn't understand	4.2.3.8	x				
Time Stamp support	4.2.3.8			x		
Record Route support	4.2.3.8			x		
Source Route:						
ALP can specify	4.2.3.8	x				1

Overrides src rt in datagram	4.2.3.8	x				
Build return route from src rt	4.2.3.8	x				
Later src route overrides	4.2.3.8		x			
Receiving ICMP Messages from IP	4.2.3.9	x				
Dest. Unreach (0,1,5) => inform ALP	4.2.3.9		x			
Dest. Unreach (0,1,5) => abort conn	4.2.3.9					x
Dest. Unreach (2-4) => abort conn	4.2.3.9		x			
Source Quench => silent discard	4.2.3.9		x			
Time Exceeded => tell ALP, don't abort	4.2.3.9		x			
Param Problem => tell ALP, don't abort	4.2.3.9		x			
Address Validation						
Reject OPEN call to invalid IP address	4.2.3.10	x				
Reject SYN from invalid IP address	4.2.3.10	x				
Silently discard SYN to bcast/mcast addr	4.2.3.10	x				
TCP/ALP Interface Services						
Error Report mechanism	4.2.4.1	x				
ALP can disable Error Report Routine	4.2.4.1		x			
ALP can specify DiffServ field for sending	4.2.4.2	x				
Passed unchanged to IP	4.2.4.2		x			
ALP can change DiffServ field during connection	4.2.4.2		x			
Pass received DiffServ field up to ALP	4.2.4.2			x		
FLUSH call	4.2.4.3			x		
Optional local IP addr parm. in OPEN	4.2.4.4	x				
-----	-----	-	-	-	-	--

FOOTNOTES: (1) "ALP" means Application-Layer program.

Author's Address

Wesley M. Eddy (editor)
 MTI Systems
 US

Email: wes@mti-systems.com

Internet Engineering Task Force
INTERNET-DRAFT
File: draft-ietf-tcpm-rto-consider-05.txt
Intended Status: Best Current Practice
Expires: September 10, 2017

M. Allman
ICSI
March 10, 2017

Retransmission Timeout Requirements

Status of this Memo

This document may not be modified, and derivative works of it may not be created, except to format it for publication as an RFC or to translate it into languages other than English.

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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

The list of current Internet-Drafts can be accessed at
<http://www.ietf.org/lid-abstracts.html>

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

This Internet-Draft will expire on September 10, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Abstract

Ensuring reliable communication often manifests in a timeout and retry mechanism. Each implementation of a retransmission timeout mechanism represents a balance between correctness and timeliness and therefore no implementation suits all situations. This document

Expires: October 10, 2017

[Page 1]

provides high-level requirements for retransmission timeout schemes appropriate for general use in the Internet. Within the requirements, implementations have latitude to define particulars that best address each situation.

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 BCP 14, RFC 2119 [RFC2119].

1 Introduction

Reliable transmission is a key property for many network protocols and applications. Our protocols use various mechanisms to achieve reliable data transmission. Often we use continuous or periodic reports from the recipient to inform the sender's notion of which pieces of data are missing and need to be retransmitted to ensure reliability. Alternatively, information coding---e.g., FEC---can be used to achieve probabilistic reliability without retransmissions. However, despite our best intentions and most robust mechanisms, the only thing we can truly depend on is the passage of time and therefore our ultimate backstop to ensuring reliability is a timeout and re-try mechanism. That is, the sender sets some expectation for how long to wait for confirmation of delivery for a given piece of data. When this time period passes without delivery confirmation the sender assumes the data was lost in transit and therefore schedules a retransmission. This process of ensuring reliability via time-based loss detection and resending lost data is commonly referred to as a "retransmission timeout (RTO)" mechanism.

Various protocols have defined their own RTO mechanisms (e.g., TCP [RFC6298], SCTP [RFC4960], SIP [RFC3261]). The specifics of retransmission timeouts often represent a particular tradeoff between correctness and responsiveness [AP99]. In other words we want to simultaneously:

- wait long enough to ensure the detection of loss is correct and therefore a retransmission is in fact needed, and
- bound the delay we impose on applications before repairing loss.

Serving both of these goals is difficult as they pull in opposite directions. I.e., towards either (a) withholding needed retransmissions too long to ensure the original transmission is truly lost or (b) not waiting long enough---to help application responsiveness---and hence sending unnecessary (often denoted "spurious") retransmissions.

We have found that even though the RTO procedure is standardized for some protocols (e.g., TCP [RFC6298]), implementations often add their own subtle imprint on the specifics of the process to tilt the

tradeoff between correctness and responsiveness in some particular way.

At this point we recognize that often these specific tweaks that deviate from standardized RTO mechanisms do not materially impact network safety. Therefore, in this document we outline a set of high-level protocol-agnostic requirements for RTO mechanisms. The intent is to provide a safe foundation on which implementations have the flexibility to instantiate mechanisms that best realize their specific goals.

2 Scope

The principles we outline in this document are protocol-agnostic and widely applicable. We make the following scope statements about the application of the requirements discussed in Section 3:

- (S.1) The requirements in this document apply only to timer-based loss detection and retransmission.

While there are a bevy of uses for timers in protocols---from rate-based pacing to connection failure detection to making congestion control decisions and beyond---these are outside the scope of this document.

- (S.2) The requirements in this document only apply to cases where loss detected via a timer is repaired by a retransmission of the original data.

Other cases are certainly possible---e.g., replacing the lost data with an updated version---but fall outside the scope of this document.

- (S.3) The requirements in this document apply only to endpoint-to-endpoint unicast communication. Reliable multicast (e.g., [RFC5740]) protocols are explicitly outside the scope of this document.

Protocols such as SCTP [RFC4960] and MP-TCP [RFC6182] that communicate in a unicast fashion with multiple specific endpoints can leverage the requirements in this document provided they track state and follow the requirements for each endpoint independently. I.e., if host A communicates with hosts B and C, A must use independent RTOs for traffic sent to B and C.

- (S.4) There are cases where state is shared across connections or flows (e.g., [RFC2140], [RFC3124]). The RTO is one piece state that is often discussed as sharable. These situations raise issues that the simple flow-oriented RTO mechanism discussed in this document does not consider (e.g., how long to preserve state between connections). Therefore, while the general principles given in Section 3 are likely applicable, sharing RTOs across flows is outside the scope of this

document.

- (S.5) The requirements in this document apply to reliable transmission, but do not assume that all data transmitted within a connection or flow is reliably sent.

E.g., a protocol like DCCP [RFC4340] could leverage the requirements in this document for the initial reliable handshake even though the protocol reverts to unreliable transmission after the handshake.

E.g., a protocol like SCTP [RFC4960] could leverage the requirements for data that is sent only "partially reliably". In this case, the protocol uses two phases for each message. In the first phase, the protocol attempts to ensure reliability and can leverage the requirements in this document. At some point the value of the data is gone and the protocol transitions to the second phase where the data is treated as unreliably transmitted and therefore the protocol will no longer attempt to repair the loss---and hence there are no more retransmissions and the requirements in this document are moot.

- (S.6) The requirements for RTO mechanisms in this document can be applied regardless of whether the RTO mechanism is the sole loss repair strategy or works in concert with other mechanisms.

E.g., for a simple protocol like UDP-based DNS [] a timeout and re-try mechanism is likely to act alone to ensure reliability.

E.g., within a complex protocol like TCP or SCTP we have designed methods to detect and repair loss based on explicit endpoint state sharing [RFC2018,RFC4960,RFC6675]. These mechanisms are preferred over the RTO as they are often more timely and precise than the coarse-grained RTO. In these cases, the RTO becomes a last resort when the more advanced mechanisms fail.

3 Requirements

We now list the requirements that apply when designing retransmission timeout (RTO) mechanisms.

- (1) In the absence of any knowledge about the latency of a path, the RTO MUST be conservatively set to no less than 1 second.

This requirement ensures two important aspects of the RTO. First, when transmitting into an unknown network, retransmissions will not be sent before an ACK would reasonably be expected to arrive and hence possibly waste scarce network resources. Second, as noted below, sometimes retransmissions can lead to ambiguities in assessing the latency of a network

path. Therefore, it is especially important for the first latency sample to be free of ambiguities such that there is a baseline for the remainder of the communication.

The specific constant (1 second) comes from the analysis of Internet RTTs found in Appendix A of [RFC6298].

- (2) As we note above, loss detection happens when a sender does not receive delivery confirmation within an some expected period of time. We now specify four requirements that pertain to setting the length of this expectation.

Often measuring the time required for delivery confirmation is is framed as involving the "round-trip time (RTT)" of the network path as this is the minimum amount of time required to receive delivery confirmation and also often follows protocol behavior whereby acknowledgments are generated quickly after data arrives. For instance, this is the case for the RTO used by TCP [RFC6298] and SCTP [RFC4960]. However, this is somewhat mis-leading as the expected latency is better framed as the "feedback time" (FT). In other words, the expectation is not always simply a network property, but includes additional time before a sender should reasonably expect a response to a query.

For instance, consider a UDP-based DNS request from a client to a recursive resolver. When the request can be served from the resolver's cache the FT likely well approximates the network RTT between the client and resolver. However, on a cache miss the resolver will request the needed information from one or more authoritative DNS servers, which will non-trivially increase the FT compared to the RTT between the client and resolver.

Therefore, we express the following requirements in terms of FT:

- (a) In steady state the RTO SHOULD be set based on recent observations of both the FT and the variance of the FT.

In other words, the RTO should be based on a reasonable amount of time that the sender should wait for delivery confirmation before retransmitting the given data.

- (b) FT observations SHOULD be taken regularly.

Internet measurements show that taking only a single FT sample per TCP connection results in a relatively poorly performing RTO mechanism [AP99], hence this requirement that the FT be sampled continuously throughout the lifetime of communication.

The notion of "regularly" SHOULD be defined as at least once per RTT or as frequently as data is exchanged in cases where that happens less frequently than once per RTT. However, we also recognize that it may not always be practical to take an FT sample this often in all cases. Hence, this

once-per-RTT definition of "regularly" is explicitly a "SHOULD" and not a "MUST".

TCP takes an FT sample roughly once per RTT, or if using the timestamp option [RFC7323] on each acknowledgment arrival. [AP99] shows that both these approaches result in roughly equivalent performance for the RTO estimator.

- (c) FT observations MAY be taken from non-data exchanges.

Some protocols use keepalives, heartbeats or other messages to exchange control information. To the extent that the latency of these transactions mirrors data exchange, they can be leveraged to take FT samples within the RTO mechanism. Such samples can help protocols keep their RTO accurate during lulls in data transmission. However, given that these messages may not be subject to the same delays as data transmission, we do not take a general view on whether this is useful or not.

- (d) An RTO mechanism MUST NOT use ambiguous FT samples.

Assume two copies of some segment X are transmitted at times t_0 and t_1 and then at time t_2 the sender receives confirmation that X in fact arrived. In some cases, it is not clear which copy of X triggered the confirmation and hence the actual FT is either t_2-t_1 or t_2-t_0 , but which is a mystery. Therefore, in this situation an implementation MUST use Karn's algorithm [KP87,RFC6298] and use neither version of the FT sample and hence not update the RTO.

There are cases where two copies of some data are transmitted in a way whereby the sender can tell which is being acknowledged by an incoming ACK. E.g., TCP's timestamp option [RFC7323] allows for segments to be uniquely identified and hence avoid the ambiguity. In such cases there is no ambiguity and the resulting samples can update the RTO.

- (3) Each time the RTO is used to detect a loss and a retransmission is scheduled, the value of the RTO MUST be exponentially backed off such that the next firing requires a longer interval. The backoff SHOULD be removed after the successful repair of the lost data and subsequent transmission of non-retransmitted data.

A maximum value MAY be placed on the RTO. The maximum RTO MUST NOT be less than 60 seconds (a la [RFC6298]).

This ensures network safety.

- (4) Retransmissions triggered by the RTO mechanism MUST be taken as indications of network congestion and the sending rate adapted using a standard mechanism (e.g., TCP collapses the congestion window to one segment [RFC5681]).

This ensures network safety.

Exception could be made to this rule if an IETF standardized mechanism is used to determine that a particular loss is due to a non-congestion event (e.g., packet corruption). In such a case a congestion control action is not required. Additionally, RTO-triggered congestion control actions may be reversed when a standard mechanism determines that the cause of the loss was not congestion after all (e.g., [RFC5682]).

4 Discussion

We note that research has shown the tension between the responsiveness and correctness of retransmission timeouts seems to be a fundamental tradeoff in the context of TCP [AP99]. That is, making the RTO more aggressive (e.g., via changing TCP's EWMA gains, lowering the minimum RTO, etc.) can reduce the time spent waiting on needed retransmissions. However, at the same time, such aggressiveness leads to more needless retransmissions. Therefore, being as aggressive as the requirements given in the previous section allow in any particular situation may not be the best course of action because an RTO expiration carries a requirement to invoke a congestion response and hence slow transmission down.

While the tradeoff between responsiveness and correctness seems fundamental, the tradeoff can be made less relevant if the sender can detect and recover from spurious RTOs. Several mechanisms have been proposed for this purpose, such as Eifel [RFC3522], F-RTO [RFC5682] and DSACK [RFC2883,RFC3708]. Using such mechanisms may allow a data originator to tip towards being more responsive without incurring (as much of) the attendant costs of needless retransmits.

Also, note, that in addition to the experiments discussed in [AP99], the Linux TCP implementation has been using various non-standard RTO mechanisms for many years seemingly without large scale problems (e.g., using different EWMA gains than specified in [RFC6298]). Further, a number of implementations use minimum RTOs that are less than the 1 second specified in [RFC6298]. While the implication of these deviations from the standard may be more spurious retransmits (per [AP99]), we are aware of no large scale network safety issues caused by this change to the minimum RTO.

Finally, we note that while allowing implementations to be more aggressive may in fact increase the number of needless retransmissions the above requirements fail safe in that they insist on exponential backoff of the RTO and a transmission rate reduction. Therefore, providing implementers more latitude than they have traditionally been given in IETF specifications of RTO mechanisms does not somehow open the flood gates to aggressive behavior. Since there is a downside to being aggressive the incentives for proper behavior are retained in the mechanism.

5 Security Considerations

This document does not alter the security properties of retransmission timeout mechanisms. See [RFC6298] for a discussion of these within the context of TCP.

Acknowledgments

This document benefits from years of discussions with Ethan Blanton, Sally Floyd, Jana Iyengar, Shawn Ostermann, Vern Paxson, and the members of the TCPM and TCP-IMPL working groups. Ran Atkinson, Yuchung Cheng, David Black, Gorry Fairhurst, Mirja Kuhlewind, Jonathan Looney and Michael Scharf provided useful comments on a previous version of this draft.

Normative References

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

Informative References

- [AP99] Allman, M., V. Paxson, "On Estimating End-to-End Network Path Properties", Proceedings of the ACM SIGCOMM Technical Symposium, September 1999.
- [KP87] Karn, P. and C. Partridge, "Improving Round-Trip Time Estimates in Reliable Transport Protocols", SIGCOMM 87.
- [RFC2018] Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, "TCP Selective Acknowledgment Options", RFC 2018, October 1996.
- [RFC2140] Touch, J., "TCP Control Block Interdependence", RFC 2140, April 1997.
- [RFC2883] Floyd, S., Mahdavi, J., Mathis, M., and M. Podolsky, "An Extension to the Selective Acknowledgement (SACK) Option for TCP", RFC 2883, July 2000.
- [RFC3124] Balakrishnan, H., S. Seshan, "The Congestion Manager", RFC 2134, June 2001.
- [RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, June 2002.
- [RFC3522] Ludwig, R., M. Meyer, "The Eifel Detection Algorithm for TCP", RFC 3522, april 2003.
- [RFC3708] Blanton, E., M. Allman, "Using TCP Duplicate Selective Acknowledgement (DSACKs) and Stream Control Transmission Protocol (SCTP) Duplicate Transmission Sequence Numbers (TSNs) to Detect Spurious Retransmissions", RFC 3708, February 2004.
- [RFC3940] Adamson, B., C. Bormann, M. Handley, J. Macker,

- "Negative-acknowledgment (NACK)-Oriented Reliable Multicast (NORM) Protocol", November 2004, RFC 3940.
- [RFC4340] Kohler, E., M. Handley, S. Floyd, "Datagram Congestion Control Protocol (DCCP)", March 2006, RFC 4340.
- [RFC4960] Stewart, R., "Stream Control Transmission Protocol", RFC 4960, September 2007.
- [RFC5682] Sarolahti, P., M. Kojo, K. Yamamoto, M. Hata, "Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP", RFC 5682, September 2009.
- [RFC5740] Adamson, B., C. Bormann, M. Handley, J. Macker, "NACK-Oriented Reliable Multicast (NORM) Transport Protocol", November 2009, RFC 5740.
- [RFC6182] Ford, A., C. Raiciu, M. Handley, S. Barre, J. Iyengar, "Architectural Guidelines for Multipath TCP Development", March 2011, RFC 6182.
- [RFC6298] Paxson, V., M. Allman, H.K. Chu, M. Sargent, "Computing TCP's Retransmission Timer", June 2011, RFC 6298.
- [RFC6582] Henderson, T., S. Floyd, A. Gurtov, Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm", April 2012, RFC 6582.
- [RFC6675] Blanton, E., M. Allman, L. Wang, I. Jarvinen, M. Kojo, Y. Nishida, "A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP", August 2012, RFC 6675.
- [RFC7323] Borman D., B. Braden, V. Jacobson, R. Scheffenegger, "TCP Extensions for High Performance", September 2014, RFC 7323.

Authors' Addresses

Mark Allman
International Computer Science Institute
1947 Center St. Suite 600
Berkeley, CA 94704

EMail: mallman@icir.org
<http://www.icir.org/mallman>

TCPM WG
Internet Draft
Updates: 793
Intended status: Standards Track
Expires: December 2017

J. Touch
USC/ISI
Wes Eddy
MTI Systems
June 26, 2017

TCP Extended Data Offset Option
draft-ietf-tcpm-tcp-edo-08.txt

Status of this Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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

The list of current Internet-Drafts can be accessed at <http://www.ietf.org/ietf/lid-abstracts.txt>

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

This Internet-Draft will expire on December 26, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in

Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

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.....	3
2. Conventions used in this document.....	3
3. Motivation.....	3
4. Requirements for Extending TCP's Data Offset.....	4
5. The TCP EDO Option.....	4
5.1. EDO Supported.....	5
5.2. EDO Extension.....	5
5.3. The two EDO Extension variants.....	8
6. TCP EDO Interaction with TCP.....	9
6.1. TCP User Interface.....	9
6.2. TCP States and Transitions.....	9
6.3. TCP Segment Processing.....	10
6.4. Impact on TCP Header Size.....	10
6.5. Connectionless Resets.....	11
6.6. ICMP Handling.....	11
7. Interactions with Middleboxes.....	12
7.1. Middlebox Coexistence with EDO.....	12
7.2. Middlebox Interference with EDO.....	13
8. Comparison to Previous Proposals.....	14
8.1. EDO Criteria.....	14
8.2. Summary of Approaches.....	15
8.3. Extended Segments.....	16
8.4. TCPx2.....	16
8.5. LO/SLO.....	17
8.6. LOIC.....	17
8.7. Problems with Extending the Initial SYN.....	18
9. Implementation Issues.....	19
10. Security Considerations.....	20
11. IANA Considerations.....	20
12. References.....	20

12.1. Normative References.....	20
12.2. Informative References.....	20
13. Acknowledgments.....	22

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][To17]. Some of these other extensions can work in conjunction with EDO (e.g., [To17]).

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 pseudoheader [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 detect 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 indicated 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][To17].

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 NAPT, 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" [Holl].

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 [Holl]. 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:

- o Zero-cost fallback to legacy endpoints.
- o Minimal impact on middlebox compatibility.
- o 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 NAT 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/NATs).

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 [Yoll]. 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][To17].

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

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

12.2. Informative References

- [Al06] Allman, M., "TCPx2: Don't Fence Me In", draft-allman-tcp2-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.
- [Hol11] 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-tcptext-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.
- [To17] Touch, J., T. Faber, "TCP SYN Extended Option Space Using an Out-of-Band Segment", draft-touch-tcpm-tcp-syn-ext-opt-07 (work in progress), June 2017.
- [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
USC/ISI
4676 Admiralty Way
Marina del Rey, CA 90292-6695 USA

Phone: +1 (310) 448-9151
Email: touch@isi.edu

Wesley M. Eddy
MTI Systems
US

Email: wes@mti-systems.com

TCP Maintenance & Minor Extensions (tcpm)
Internet-Draft
Updates: 793, 2018, 5925, 7323 (if
approved)
Intended status: Standards Track
Expires: September 14, 2017

J. Looney
Netflix
March 13, 2017

64-bit Sequence Numbers for TCP
draft-looney-tcpm-64-bit-seqnos-00

Abstract

This draft updates RFC 793 to allow the optional use of 64-bit sequence numbers. It also updates other standards to support the extended sequence number space.

Status of This Memo

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

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

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

This Internet-Draft will expire on September 14, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1.	Introduction	2
1.1.	Design Goals	3
1.2.	Overview of Implementation	3
1.3.	Backwards Compatibility	4
1.4.	Requirements Language	4
2.	Extended Sequence Numbers	4
2.1.	The 64-bit Sequence Number Option	5
2.2.	Operation of the 64-bit Sequence Number Option	6
2.2.1.	Choice of Initial Sequence Numbers	6
2.2.2.	Negotiation of the 64-bit Sequence Number Option	7
2.2.3.	Detecting Middle Boxes	8
2.2.4.	Backwards Compatibility Mode	8
3.	Changes to Other Features	9
3.1.	Window Size	9
3.2.	SACK Blocks	9
3.2.1.	32-bit SACK Blocks	9
3.2.2.	64-bit SACK Blocks	10
3.3.	TCP Authentication Option	11
3.4.	Other Features	12
4.	Implementation Considerations	12
5.	Acknowledgements	13
6.	IANA Considerations	13
7.	Security Considerations	14
7.1.	Attacks Due to Sequence-Number Guessing	14
7.2.	Downgrade Attacks	14
7.3.	Denial-of-Service Attacks	15
7.4.	32-bit Sequence Numbers	15
8.	References	15
8.1.	Normative References	15
8.2.	Informative References	16
Appendix A.	Design Choices	16
A.1.	Detecting Middle Boxes	16
A.2.	SACK Blocks	17
Author's Address	17

1. Introduction

RFC 793 [RFC0793] specifies the sequence number space as a 32-bit space. This means that the sequence number space will wrap in 2^{32} bytes. On a 10-Gb/s network, this can occur in approximately 3.5 seconds. On a 100-Gb/s network, this can occur in approximately 350 milliseconds. While sequence number wrapping is a basic feature of TCP, the specified wrapping mechanism only supports having a theoretical maximum of 2^{31} bytes outstanding at any given time. Additionally, when you are re-using sequence number space in such a short timeframe, it is unclear that the existing mechanisms for

detecting duplicate packets will be sufficient. To practically support these very high-speed networks, it is necessary to expand the sequence number space.

In addition to the base TCP specification, a number of other specifications have made assumptions about sequence numbers being 32 bits long. This document updates some of those specifications and provides guidance on interaction with other specifications.

1.1. Design Goals

This document assumes the following design goals:

- o Support 64-bit sequence numbers.
- o Maintain the existing header format.
- o Maintain backwards compatibility with TCP implementations (including middle boxes) that only support 32-bit sequence numbers.
- o Require minimal changes for any features that assume 32-bit sequence numbers.
- o Use minimal TCP option space.

1.2. Overview of Implementation

This document specifies that the least significant 32 bits of the sequence number will continue to be carried in the Sequence Number and Acknowledgment Number fields of the standard TCP header. The most significant 32 bits will be carried in a new TCP option.

This mechanism provides an easy way to negotiate the option on startup: hosts that understand 64-bit sequence numbers can include the option with the SYN. If the other host does not understand the 64-bit Sequence Number Option, it will ignore the option and use the 32-bit sequence number already contained in the standard TCP header. When the initiating host receives a SYN/ACK that does not contain the 64-bit sequence number option, it simply reverts to normal 32-bit operation.

This method of negotiation and operation bears some similarity to the TCP Timestamp Option [RFC7323], which has been widely deployed without evident problems.

The 64-bit acknowledgment number is divided in the same way. For completeness, the acknowledgment number logical divisions are shown in Figure 2.

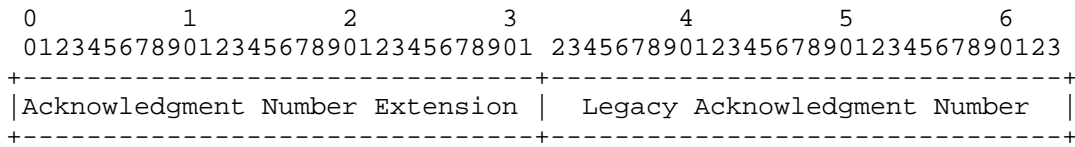


Figure 2: Logical division of a 64-bit acknowledgment number

In Figure 2, the least-significant 32 bits of the acknowledgment number are labeled the "Legacy Acknowledgment Number". This is the portion of the acknowledgment number that is stored in the Acknowledgment Number field of the standard TCP header defined in [RFC0793]. In Figure 2, the most-significant 32 bits of the acknowledgment number are labeled the "Acknowledgment Number Extension". This is the portion of the acknowledgment number that is stored in the 64-bit Sequence Number Option, which is defined in this document.

2.1. The 64-bit Sequence Number Option

The 64-bit Sequence Number Option is used to carry the most-significant 32 bits of the 64-bit sequence number information. It is also used to signal support for 64-bit sequence numbers in TCP segments with the SYN flag set.

The 64-bit Sequence Number Option will use TCP Option Kind TBD1. Its general form is shown in Figure 3.

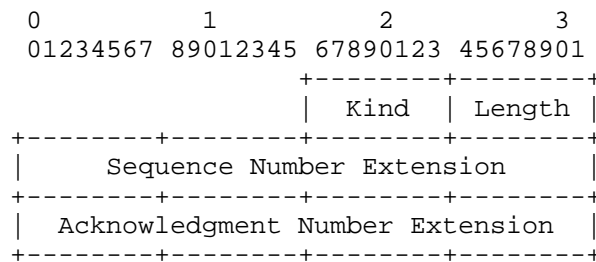


Figure 3: The 64-bit Sequence Number Option

Prior to standardization action, implementations should use the mechanism described in [RFC6994] to encode the option. IANA has reserved experiment ID (ExID) TBD2 for the option described in this document. This option format is shown in Figure 4.

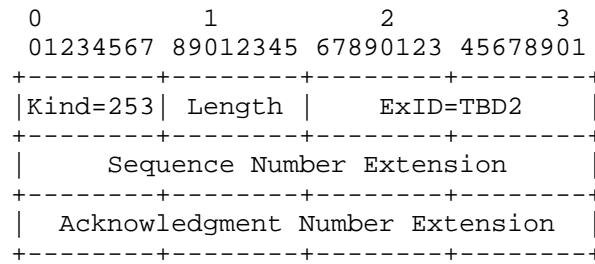


Figure 4: The 64-bit Sequence Number Option with ExID

In both cases, the fields are described in more detail below:

Length

The total length (in octets) of the option (including Kind, Length, and, if applicable, ExID), as specified in [RFC0793].

Sequence Number Extension

The most-significant 32 bits of the 64-bit sequence number.

Acknowledgment Number Extension

For a segment with the ACK flag set, this field contains the most-significant 32 bits of the 64-bit sequence number. If the ACK flag is not set, this field is omitted (and, consequently, the option is 4 octets shorter).

2.2. Operation of the 64-bit Sequence Number Option

In order to use 64-bit sequence numbers, it is necessary for both hosts to negotiate the use of 64-bit sequence numbers. Further, it is necessary to ensure that no middlebox that is unaware of 64-bit sequence numbers is going to modify sequence number information. This section describes the initial negotiation to satisfy these parameters.

2.2.1. Choice of Initial Sequence Numbers

In order to detect when a middlebox has modified the initial sequence numbers (ISNs) in the three-way handshake, each host must choose an ISN such that the Sequence Number Extension is the bitwise inverse of the Legacy Sequence Number. In C pseudo-code:

```
sequence_number_extension = ~(legacy_sequence_number);
```

This property is only a restriction on a choice of ISN. Subsequent to the selection of an ISN, 64-bit sequence numbers behave as normal 64-bit numbers.

2.2.2. Negotiation of the 64-bit Sequence Number Option

When a host ("the client") desires to use 64-bit sequence numbers for a TCP connection it is initiating, it includes the 64-bit Sequence Number Option in the initial segment. It places the least-significant 32 bits of the initial sequence number (ISN) in the Sequence Number field of the TCP header. It places the most-significant 32 bits of the ISN in the Sequence Number Extension field of the 64-bit Sequence Number Option.

When a host ("the server") receives a request to initiate a TCP connection (that is, a segment with the SYN flag set and the ACK flag not set) and the segment contains a valid 64-bit Sequence Number Option, the server MAY choose to use 64-bit sequence numbers for that TCP connection. If the server chooses to use 64-bit sequence numbers for that connection, the server includes the 64-bit sequence number option in its reply (that is, a segment with both the SYN and ACK flags set). The server MUST NOT include a 64-bit Sequence Number Option unless the client included the 64-bit Sequence Number Option in its request to initiate a TCP connection.

When the client receives the initial reply (that is, a segment with both the SYN and ACK flags set), it checks for a valid 64-bit Sequence Number Option. If it finds a valid 64-bit Sequence Number Option, it MUST include the 64-bit Sequence Number Option on all subsequent segments it sends for this connection.

When the server receives an acknowledgement to its initial segment, it checks for a valid 64-bit Sequence Number Option. If it finds a valid 64-bit Sequence Number Option, it MUST include the 64-bit Sequence Number Option on all subsequent segments it sends for this connection.

For purposes of this section, a 64-bit Sequence Number Option is considered "valid" if (and only if):

- o If the segment's SYN flag is set, the Sequence Number Extension must be the bitwise inverse of the Legacy Sequence Number.
- o If the ACK flag is set, the full 64-bit acknowledgment number exactly matches the expected value.

A host is said to have negotiated to use 64-bit sequence numbers if it has sent a 64-bit Sequence Number Option in the first segment it sent to the remote host and the first reply it received from the remote host contained a valid 64-bit Sequence Number Option.

If a host successfully negotiates the use of 64-bit sequence numbers, the host proceeds using 64-bit sequence numbers for the remainder of the session. If a host fails to successfully negotiate the use of 64-bit sequence numbers, the host uses backwards compatibility mode (see Section 2.2.4).

2.2.3. Detecting Middle Boxes

If a middle box is present which is modifying sequence numbers or proxying TCP connections, and that middle box does not support 64-bit sequence numbers, it is probable that either the ISN will not follow the rule specified in Section 2.2.1 or the Acknowledgment Number will not match the expected values. (The probability that these will exactly match accidentally is approximately 1 in 2^{32} . And, it is hard to conceive of a reasonable scenario where the 32-bit sequence numbers will exactly match, the first three segments will all also contain valid 64-bit Sequence Number Options, and yet the two sides will be unable to communicate using 64-bit sequence numbers.) That is why both sides MUST follow the validation rules specified in Section 2.2.2 for the first first three packets in the session (the so-called "three-way handshake"). And, this is also why both sides MUST fallback to using 32-bit sequence numbers if an invalid 64-bit Sequence Number Option is detected in one of the first three frames.

2.2.4. Backwards Compatibility Mode

If a host finds a missing or invalid 64-bit Sequence Number Option in one of the first three segments of a connection (the so-called "three-way handshake"), it MUST process the segment using 32-bit sequence numbers. Specifically, it ignores any 64-bit Sequence Number Option and only pays attention to the 32-bit Sequence Number and Acknowledgement Number fields found in the standard TCP header. Additionally, the host only considers the Legacy Sequence Number portion of the 64-bit sequence number and/or acknowledgement number it stored for the session. If the host finds that the segment is still not valid (e.g. the Acknowledgment Number does not match the expected value), it ignores the segment. (NOTE: This specifically means that the segment does NOT determine whether the host has successfully negotiated, or failed to negotiate, the use of 64-bit sequence numbers on the session.)

However, if the host finds that the segment is valid when processed using 32-bit sequence numbers, the 64-bit sequence number negotiation has failed and the host MUST proceed for the rest of the session using only 32-bit sequence numbers. In this case, it MUST NOT send the 64-bit Sequence Number Option on any further segments for that connection.

If a host has NOT successfully negotiated to use 64-bit sequence numbers for a particular connection and it receives a 64-bit Sequence Number Option in a TCP segment for that connection, it MUST treat the segment as if it contained an out-of-window sequence number.

If a host has successfully negotiated to use 64-bit sequence numbers for a particular connection and it receives a segment without a 64-bit Sequence Number Option, it MUST treat the segment as if it contained an out-of-window sequence number.

3. Changes to Other Features

Over time, other features have built upon the base TCP protocol specification. Many, if not all, of these features have assumed the existence of 32-bit sequence numbers. This document updates some of the features. It also provides a general rule for the operation of other features.

3.1. Window Size

[RFC7323] specifies the Window Scale Option. It specifies a maximum window shift of 14. This document updates [RFC7323] by specifying that the maximum window shift is 46 if the hosts successfully negotiate using 64-bit sequence numbers for a connection. If the 64-bit sequence number negotiation fails, both hosts must enforce the maximum window shift of 14 specified by [RFC7323].

3.2. SACK Blocks

[RFC2018] defines a way to acknowledge receipt of out-of-order segments. [RFC2018] specifies that the segments are defined by 32-bit sequence numbers. This document updates the way SACK blocks defined in [RFC2018] are interpreted when used on 64-bit environments. It also defines a new option used to hold 64-bit SACK blocks.

3.2.1. 32-bit SACK Blocks

When a host has successfully negotiated the use of 64-bit sequence numbers on a session and has also negotiated the use of SACK as described in [RFC2018], the host may append a TCP SACK option as defined in [RFC2018]. When these options are used, the sequence numbers in the option are interpreted as follows: the Acknowledgment Number Extension from the 64-bit Sequence Number Option is used as the most-significant 32 bits of the 64-bit sequence numbers, while the sequence numbers from the TCP SACK option are used as the least-significant 32 bits of the 64-bit sequence numbers. Otherwise, the operation of the TCP SACK option remains unchanged.

3.2.2. 64-bit SACK Blocks

When a host has successfully negotiated the use of 64-bit sequence numbers on a session and has also negotiated the use of SACK as described in [RFC2018], the host may append a 64-bit SACK Option.

The 64-bit SACK Option will use TCP Option Kind TBD3. Its general form is shown in Figure 5.

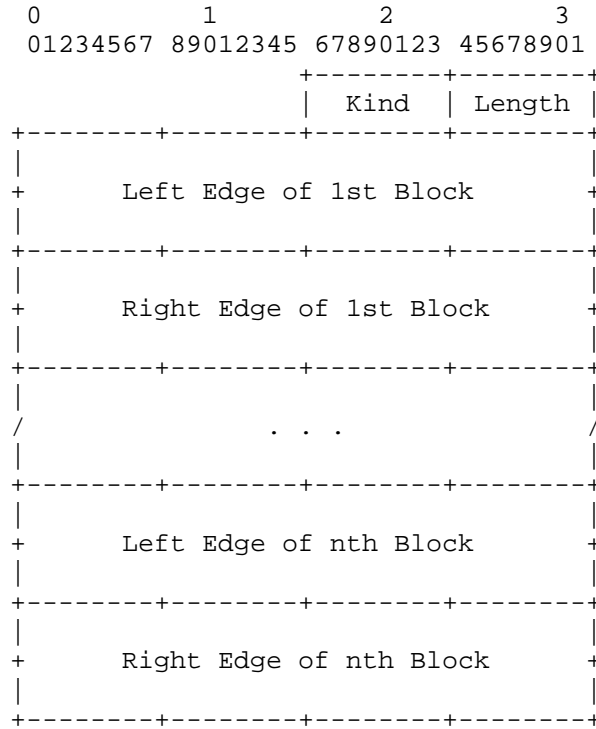


Figure 5: The 64-bit SACK Option

Prior to standardization action, implementations should use the mechanism described in [RFC6994] to encode the option. IANA has reserved experiment ID (ExID) TBD4 for the option described in this document. This option format is shown in Figure 6.

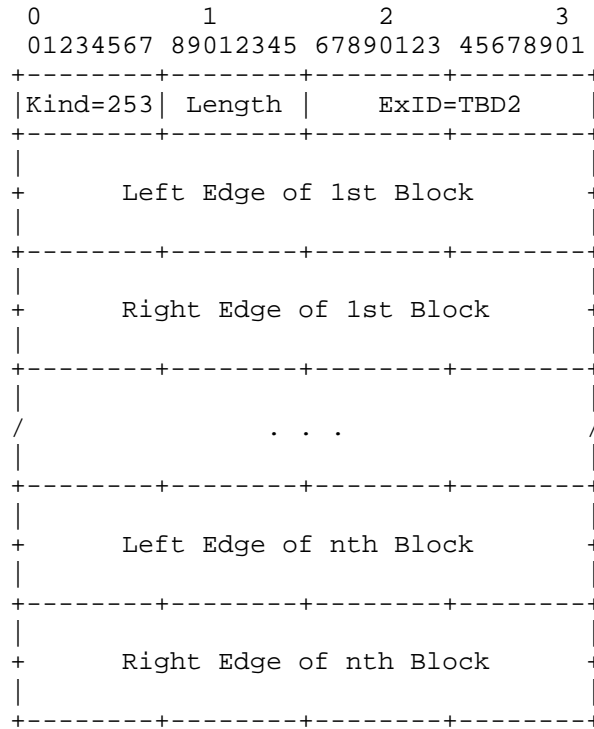


Figure 6: The 64-bit SACK Option with ExID

The meaning of the option fields, and the operation of the option, is unchanged from the TCP SACK option described in [RFC2018], except that the sequence numbers are 64-bit values in network byte order.

3.3. TCP Authentication Option

[RFC5925] defines the TCP Authentication Option. The TCP Authentication Option uses sequence numbers in two places: the Key Derivation Function (KDF) context and the data input to the Message Authentication Code (MAC) algorithm.

For purposes of the KDF context, this document updates [RFC5925] to specify that the Source ISN and Dest. ISN fields (shown in Figure 7 of [RFC5925]) are defined to be the least-significant 32 bits of the initial sequence numbers.

For purposes of the input to the Message Authentication Code (MAC) algorithm, this document updates [RFC5925] to specify that the Sequence Number Extension field is the Sequence Number Extension field from the 64-bit Sequence Number Option, if the option is

present, in any packet that does not carry the SYN flag. Otherwise, the Sequence Number Extension field is calculated as specified in [RFC5925].

Note that the Sequence Number Extension field will always be formulated as specified in [RFC5925] for the first two packets of the so-called "three-way handshake". This ensures that hosts will be able to correctly calculate MACs whether or not they support 64-bit sequence numbers.

3.4. Other Features

Anywhere that another RFC specifies the use of sequence numbers without specifying the way 64-bit sequence numbers should be handled, the RFC shall be interpreted as using the least-significant 32 bits of the sequence number.

4. Implementation Considerations

During the 64-bit sequence number negotiation, it is important for security purposes (as described in Section 7.3) that the server check the third packet of the "three-way handshake" when determining whether the connection has negotiated to use 64-bit sequence numbers. If another in-sequence packet is received prior to the third packet of the "three-way handshake", it must either be discarded or queued for processing after the third packet of the "three-way handshake" is received.

It may be useful to provide a way for applications to know whether a given connection uses 32-bit or 64-bit sequence numbers. It may also be useful to provide a way for applications to force the use of 32-bit or 64-bit sequence numbers.

It will be essential to properly handle 32-bit and 64-bit sequence numbers concurrently for different connections. This will require providing two sets of arithmetic and comparison functions. For various reasons, it probably makes sense to store the data as a union of a single 64-bit value and a two-member array of 32-bit values.

Due to the limited option space, it may be impossible to deploy this feature concurrently with some other features on a given connection. This limitation may change if the option space is expanded by a future standardization change. However, implementers should pay attention to the possible combinations of options and order them in such a way to fit the maximum number of options in a single segment. Further, implementations will need to prioritize which features actually appear in the option space if they will not all fit.

Careful consideration will need to be paid to various offload technologies, such as TCP segmentation offload (TSO) or large receive offload (LRO). If the network interface card (NIC) drivers or hardware do not support 64-bit sequence numbers, the endpoint MUST NOT try to use 64-bit sequence numbers. Otherwise, sessions may not work correctly in practice, even if they appear to work correctly in small-scale tests.

Implementations must ensure that the least-significant 32 bits of 64-bit initial Sequence Numbers (ISNs) must serve as sufficiently random 32-bit ISNs. (See Section 7.4.)

5. Acknowledgements

Jana Iyengar, Randall Stewart, and Michael Tuexen provided valuable feedback on this document. Michael Tuexen suggested the mechanism that currently appears in Section 2.2.1.

6. IANA Considerations

IANA has assigned an option code value of TBD1 to the 64-bit Sequence Number Option (defined in Section 2.1) and an option code value of TBD3 to the 64-bit SACK Option (defined in Section 3.2.2) from the TCP Option Kind Numbers space defined in Section 9.3 of RFC 2780 [RFC2780].

[Note to editor: I think this paragraph and the following table can be removed.]The requested options are summarized below:

Value	Description	Reference
TBD1	64-bit Sequence Number	[RFCXXXX]
TBD3	64-bit SACK	[RFCXXXX]

IANA has assigned an identifier value of TBD2 to the 64-bit Sequence Number experiment and an identifier value of TBD4 to the 64-bit SACK experiment from the TCP Experimental Option Experiment Identifiers space defined in Section 8 of RFC 6994 [RFC6994] [NOTE: If this is standardized with an option number, the experimental IDs should be deprecated, which will require change to this text.]

[Note to editor: I think this paragraph and the following table can be removed.]The assigned ExIDs are summarized below:

Value	Description	Reference
TBD2	64-bit Sequence Number	[draft-looney-tcpm-64-bit-seqnos-00]
TBD4	64-bit SACK	[draft-looney-tcpm-64-bit-seqnos-00]

7. Security Considerations

The security properties of TCP are largely unchanged (at least in a negative way) by 64-bit sequence numbers. However, a few things are worth discussing.

7.1. Attacks Due to Sequence-Number Guessing

With 32-bit sequence numbers and the maximum window shift, an attacker has approximately a 25% chance of accurately guessing an in-window Sequence Number. If a host checks for both the acceptability of Sequence Numbers and Acknowledgment Numbers prior to acting on a segment, in the worst-case scenario (where the full window size is in flight, allowing for a full window size worth of acceptable Acknowledgment Numbers), this allows a 6.25% chance of accurately guessing a combination of in-window Sequence Number and acceptable Acknowledgment Number.

Because this document specifies a maximum window shift that is 32 bits larger than the maximum window shift used for 32-bit sequence numbers, these security properties are essentially unchanged with 64-bit sequence numbers. (The major change is that an out-of-band attacker may not be able to guess whether a connection uses 64-bit sequence numbers. This may require that they try both 32-bit and 64-bit sequence number semantics, decreasing the chance that they would accurately guess appropriate sequence numbers.)

However, if you compare the use of 32-bit and 64-bit sequence numbers with the same amount of outstanding traffic and the same window size, the chance of guessing acceptable sequence numbers is much smaller with 64-bit sequence numbers than 32-bit sequence numbers.

7.2. Downgrade Attacks

A man-in-the-middle (for example, a middlebox or proxy) can conduct a downgrade attack. This is actually a feature, as it allows two endpoints that understand 64-bit sequence numbers to communicate through a middlebox or proxy that does not understand 64-bit sequence numbers. However, it is important that operators be cognizant of the differing performance and security properties of 32-bit and 64-bit

sequence numbers. It may be appropriate to provide a mechanism for applications to require the use of 64-bit sequence numbers (and reset a session that cannot be established with 64-bit sequence numbers).

7.3. Denial-of-Service Attacks

This mechanism introduces one additional denial-of-service attack possibility. Assume a session where both sides have sent and received valid 64-bit Sequence Number Options in the SYN segments. If an attacker correctly guesses the appropriate Sequence Number and Acknowledgment Number to use in the third packet of the so-called "three-way handshake" and they can inject a packet with the correct Sequence Number and Acknowledgment Number without a 64-bit Sequence Number Option and ensure the server receives the spoofed packet prior to the valid packet, this will prevent communication between the two hosts. The server will use 32-bit sequence numbers for the session, while the client will use 64-bit sequence numbers for the session. However, the requirement that the server must verify the actual third packet of the "three-way handshake" (and not merely some in-window segment) requires that the attacker EXACTLY guess both the 32-bit Legacy Sequence Number and the 32-bit Legacy Acknowledgment Number. With completely random sequence numbers, the chance of doing this is 1 in 2^{64} .

7.4. 32-bit Sequence Numbers

Because a session that is started with 64-bit sequence numbers may fallback to using 32-bit sequence numbers, implementations MUST choose ISNs such that the least-significant 32 bits of the ISN must be at least as random as the 32-bit ISNs that the system uses for connections that only support 32-bit sequence numbers.

Further, the mechanism chosen to detect middleboxes results in only 2^{32} possible 64-bit ISNs. This provides the same level of security provided with 32-bit sequence numbers.

8. References

8.1. 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>>.
- [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>>.

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

8.2. Informative References

- [RFC2780] Bradner, S. and V. Paxson, "IANA Allocation Guidelines For Values In the Internet Protocol and Related Headers", BCP 37, RFC 2780, DOI 10.17487/RFC2780, March 2000, <<http://www.rfc-editor.org/info/rfc2780>>.

Appendix A. Design Choices

This section attempts to document the reasoning behind some of the design choices.

A.1. Detecting Middle Boxes

An earlier version of this draft specified a different mechanism for detecting middlebox changes: a checksum of the 64-bit Sequence Number and Acknowledgment Number. This had the benefit of allowing the full 64-bit sequence number to be random. However, it had the negative effects of requiring an additional two bytes of option space and requiring additional processing on input and output. Michael Tuexen suggested the mechanism that currently appears in Section 2.2.1.

The mechanism that currently appears in Section 2.2.3 may still fail to detect a middlebox in one case. If there is a middlebox (such as a "transparent proxy") that passes TCP segments unchanged between the client and server, rewriting only IP addresses, this mechanism will not detect such a middlebox. However, it is not really necessary to detect such a middlebox: if the middlebox literally leaves the TCP portion of the packet unchanged, it should be perfectly acceptable to use 64-bit sequence numbers.

A.2. SACK Blocks

An earlier version of this draft reused the existing TCP SACK option and specified that the option should contain sequence numbers of the same length as the sequence numbers in use for the connection. However, this was suboptimal for two reasons. First, a middle box might misinterpret the meaning of the 64-bit sequence numbers. Second, it always required the use of 64-bit values. The current mechanism means that the existing TCP SACK option will always contain 32-bit values. This mechanism also allows the use of 32-bit values instead of full 64-bit values in some cases. However, this may still suffer from being too complex.

Author's Address

Jonathan Looney
Netflix
100 Winchester Circle
Los Gatos, CA 95032
USA

Email: jtl.ietf@gmail.com

TCPM WG
Internet Draft
Intended status: Informational
Expires: July 2017

J. Touch
USC/ISI
M. Welzl
S. Islam
University of Oslo
J. You
Huawei
January 12, 2017

TCP Control Block Interdependence
draft-touch-tcpm-2140bis-02.txt

Status of this Memo

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

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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

The list of current Internet-Drafts can be accessed at
<http://www.ietf.org/ietf/lid-abstracts.txt>

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

This Internet-Draft will expire on July 12, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Abstract

This memo describes interdependent TCP control blocks, where part of the TCP state is shared among similar concurrent or consecutive connections. TCP state includes a combination of parameters, such as connection state, current round-trip time estimates, congestion control information, and process information. Most of this state is maintained on a per-connection basis in the TCP Control Block (TCB), but implementations can (and do) share certain TCB information across connections to the same host. Such sharing is intended to improve overall transient transport performance, while maintaining backward-compatibility with existing implementations. The sharing described herein is limited to only the TCB initialization and so has no effect on the long-term behavior of TCP after a connection has been established.

Table of Contents

1. Introduction.....	3
2. Conventions used in this document.....	3
3. Terminology.....	4
4. The TCP Control Block (TCB).....	4
5. TCB Interdependence.....	5
6. An Example of Temporal Sharing.....	5
7. An Example of Ensemble Sharing.....	8
8. Compatibility Issues.....	10
9. Implications.....	12
10. Implementation Observations.....	14
11. Security Considerations.....	15
12. IANA Considerations.....	16
13. References.....	17
13.1. Normative References.....	17

13.2. Informative References.....	17
14. Acknowledgments.....	19

1. Introduction

TCP is a connection-oriented reliable transport protocol layered over IP [RFC793]. Each TCP connection maintains state, usually in a data structure called the TCP Control Block (TCB). The TCB contains information about the connection state, its associated local process, and feedback parameters about the connection's transmission properties. As originally specified and usually implemented, most TCB information is maintained on a per-connection basis. Some implementations can (and now do) share certain TCB information across connections to the same host. Such sharing is intended to lead to better overall transient performance, especially for numerous short-lived and simultaneous connections, as often used in the World-Wide Web [Be94],[Br02].

This document discusses TCB state sharing that affects only the TCB initialization, and so has no effect on the long-term behavior of TCP after a connection has been established. Path information shared across SYN destination port numbers assumes that TCP segments having the same host-pair experience the same path properties, irrespective of TCP port numbers. The observations about TCB sharing in this document apply similarly to any protocol with congestion state, including SCTP [RFC4960] and DCCP [RFC4340], as well as for individual subflows in Multipath TCP [RFC6824].

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 significance described in RFC 2119.

In this document, the characters ">>" preceding an indented line(s) indicates a statement using the key words listed above. This convention aids reviewers in quickly identifying or finding the portions of this RFC covered by these keywords.

3. Terminology

Host - a source or sink of TCP segments associated with a single IP address

Host-pair - a pair of hosts and their corresponding IP addresses

Path - an Internet path between the IP addresses of two hosts

4. The TCP Control Block (TCB)

A TCB describes the data associated with each connection, i.e., with each association of a pair of applications across the network. The TCB contains at least the following information [RFC793]:

- Local process state
 - pointers to send and receive buffers
 - pointers to retransmission queue and current segment
 - pointers to Internet Protocol (IP) PCB
- Per-connection shared state
 - macro-state
 - connection state
 - timers
 - flags
 - local and remote host numbers and ports
 - TCP option state
 - micro-state
 - send and receive window state (size*, current number)
 - round-trip time and variance
 - cong. window size (snd_cwnd)*
 - cong. window size threshold (ssthresh)*
 - max window size seen*
 - sendMSS#
 - MMS_S#
 - MMS_R#
 - PMTU#
 - round-trip time and variance#

The per-connection information is shown as split into macro-state and micro-state, terminology borrowed from [Co91]. Macro-state describes the finite state machine; we include the endpoint numbers and components (timers, flags) used to help maintain that state. Macro-state describes the protocol for establishing and maintaining shared state about the connection. Micro-state describes the protocol after a connection has been established, to maintain the reliability and congestion control of the data transferred in the connection.

We further distinguish two other classes of shared micro-state that are associated more with host-pairs than with application pairs. One class is clearly host-pair dependent (#, e.g., MSS, MMS, PMTU, RTT), and the other is host-pair dependent in its aggregate (*, e.g., congestion window information, current window sizes, etc.).

5. TCB Interdependence

There are two cases of TCB interdependence. Temporal sharing occurs when the TCB of an earlier (now CLOSED) connection to a host is used to initialize some parameters of a new connection to that same host, i.e., in sequence. Ensemble sharing occurs when a currently active connection to a host is used to initialize another (concurrent) connection to that host.

6. An Example of Temporal Sharing

The TCB data cache is accessed in two ways: it is read to initialize new TCBS and written when more current per-host state is available. New TCBS are initialized using context from past connections as follows:

TEMPORAL SHARING - TCB Initialization

Safe?	Cached TCB	New TCB
yes	old_MMS_S	old_MMS_S or not cached
yes	old_MMS_R	old_MMS_R or not cached
yes	old_sendMSS	old_sendMSS
yes	old_PMTU	old_PMTU
TBD	old_RTT	old_RTT
TBD	old_RTTvar	old_RTTvar
varies	old_option	(option specific)
TBD	old_ssthresh	old_ssthresh
TBD	old_snd_cwnd	old_snd_cwnd

Table entries indicate which are considered to be safe to share temporally. The other entries are discussed in section 8.

Most cached TCB values are updated when a connection closes. The exceptions are MMS_R and MMS_S, which are reported by IP [RFC1122], PMTU which is updated after Path MTU Discovery [RFC1191][RFC1981][RFC4821], and sendMSS, which is updated if the MSS option is received in the TCP SYN header.

Sharing sendMSS information affects only data in the SYN of the next connection, because sendMSS information is typically included in most TCP SYN segments. Caching PMTU can accelerate the efficiency of PMTUD, but can also result in black-holing until corrected if in error. Caching MMS_R and MMS_S may be of little direct value as they are reported by the local IP stack anyway.

[TBD - complete this section with details for TFO and other options whose state may, must, or must not be shared] The way in which other TCP option state can be shared depends on the details of that option. E.g., TFO state includes the TCP Fast Open Cookie [RFC7413] or, in case TFO fails, a negative TCP Fast Open response (from [RFC 7413]: "The client MUST cache negative responses from the server in order to avoid potential connection failures. Negative responses include the server not acknowledging the data in the SYN, ICMP error messages, and (most importantly) no response (SYN-ACK) from the server at all, i.e., connection timeout."). TFOinfo is cached when a connection is established.

Other TCP option state might not be as readily cached. E.g., TCP-AO [RFC5925] success or failure between a host pair for a single SYN destination port might be usefully cached. TCP-AO success or failure to other SYN destination ports on that host pair is never useful to cache because TCP-AO security parameters can vary per service.

The table below gives an overview of option-specific information that is considered safe to share.

TEMPORAL SHARING - Option info

Cached	New
old_TFO_Cookie	old_TFO_Cookie
old_TFO_Failure	old_TFO_Failure

TEMPORAL SHARING - Cache Updates

Safe?	Cached TCB	Current TCB	when?	New Cached TCB
yes	old_MMS_S	curr_ MMS_S	OPEN	curr MMS_S
yes	old_MMS_R	curr_ MMS_R	OPEN	curr_MMS_R
yes	old_sendMSS	curr_sendMSS	MSSopt	curr_sendMSS
yes	old_PMTU	curr_PMTU	PMTUD	curr_PMTU
TBD	old_RTT	curr_RTT	CLOSE	merge(curr,old)
TBD	old_RTTvar	curr_RTTvar	CLOSE	merge(curr,old)
varies	old_option	curr option	ESTAB	(depends on option)
TBD	old_ssthresh	curr_ssthresh	CLOSE	merge(curr,old)
TBD	old_snd_cwnd	curr_snd_cwnd	CLOSE	merge(curr,old)

Caching PMTU and sendMSS is trivial; reported values are cached, and the most recent values are used. The cache is updated when the MSS option is received in a SYN or after PMTUD (i.e., when an ICMPv4 Fragmentation Needed [RFC1191] or ICMPv6 Packet Too Big message is received [RFC1981] or the equivalent is inferred, e.g. as from PLPMTUD [RFC4821]), respectively, so the cache always has the most recent values from any connection. For sendMSS, the cache is consulted only at connection establishment and not otherwise updated, which means that MSS options do not affect current connections. The default sendMSS is never saved; only reported MSS values update the cache, so an explicit override is required to reduce the sendMSS. There is no particular benefit to caching MMS_S and MMS_R as these are reported by the local IP stack.

TCP options are copied or merged depending on the details of each option. E.g., TFO state is updated when a connection is established and read before establishing a new connection.

RTT values are updated by a more complicated mechanism [RFC1644][Ja86]. Dynamic RTT estimation requires a sequence of RTT measurements. As a result, the cached RTT (and its variance) is an average of its previous value with the contents of the currently active TCB for that host, when a TCB is closed. RTT values are updated only when a connection is closed. The method for merging old and current values needs to attempt to reduce the transient for new

connections. [THESE MERGE FUNCTIONS NEED TO BE SPECIFIED, considering e.g. [DM16] - TBD].

The updates for RTT, RTTvar and ssthresh rely on existing information, i.e., old values. Should no such values exist, the current values are cached instead.

TEMPORAL SHARING - Option info Updates

Cached	Current	when?	New Cached
old_TFO_Cookie	old_TFO_Cookie	ESTAB	old_TFO_Cookie
old_TFO_Failure	old_TFO_Failure	ESTAB	old_TFO_Failure

7. An Example of Ensemble Sharing

Sharing cached TCB data across concurrent connections requires attention to the aggregate nature of some of the shared state. For example, although MSS and RTT values can be shared by copying, it may not be appropriate to copy congestion window or ssthresh information (see section 8 for a discussion of congestion window or ssthresh sharing).

ENSEMBLE SHARING - TCB Initialization

Safe?	Cached TCB	New TCB
yes	old_MMS_S	old_MMS_S
yes	old_MMS_R	old_MMS_R
yes	old_sendMSS	old_sendMSS
yes	old_PMTU	old_PMTU
TBD	old_RTT	old_RTT
TBD	old_RTTvar	old_RTTvar
TBD	old_option	(option-specific)

Table entries indicate which are considered to be safe to share across an ensemble. The other entries are discussed in section 8.

The table below gives an overview of option-specific information that is considered safe to share.

ENSEMBLE SHARING - Option info

Cached	New
old_TFO_Cookie	old_TFO_Cookie
old_TFO_Failure	old_TFO_Failure

ENSEMBLE SHARING - Cache Updates

Safe?	Cached TCB	Current TCB	when?	New Cached TCB
yes	old_MMS_S	curr_MMS_S	OPEN	curr_MMS_S
yes	old_MMS_R	curr_MMS_R	OPEN	curr_MMS_R
yes	old_sendMSS	curr_sendMSS	MSSopt	curr_sendMSS
yes	old_PMTU	curr_PMTU	PMTUD /PLPMTUD	curr_PMTU
TBD	old_RTT	curr_RTT	update	rtt_update(old,cur)
TBD	old_RTTvar	curr_RTTvar	update	rtt_update(old,cur)
varies	old_option	curr option	(depends)	(option specific)

For ensemble sharing, TCB information should be cached as early as possible, sometimes before a connection is closed. Otherwise, opening multiple concurrent connections may not result in TCB data sharing if no connection closes before others open. The amount of work involved in updating the aggregate average should be minimized, but the resulting value should be equivalent to having all values measured within a single connection. The function "rtt_update" in the ensemble sharing table indicates this operation, which occurs whenever the RTT would have been updated in the individual TCP connection. As a result, the cache contains the shared RTT variables, which no longer need to reside in the TCB [Ja86].

Congestion window size and ssthresh aggregation are more complicated in the concurrent case. When there is an ensemble of connections, we

need to decide how that ensemble would have shared these variables, in order to derive initial values for new TCBS.

ENSEMBLE SHARING - Option info Updates

Cached	Current	when?	New Cached
old_TFO_Cookie	old_TFO_Cookie	ESTAB	old_TFO_Cookie
old_TFO_Failure	old_TFO_Failure	ESTAB	old_TFO_Failure

Any assumption of this sharing can be incorrect, including this one, because identical endpoint address pairs may not share network paths. In current implementations, new congestion windows are set at an initial value of 4-10 segments [RFC3390][RFC6928], so that the sum of the current windows is increased for any new connection. This can have detrimental consequences where several connections share a highly congested link.

There are several ways to initialize the congestion window in a new TCB among an ensemble of current connections to a host, as shown below. Current TCP implementations initialize it to four segments as standard [rfc3390] and 10 segments experimentally [RFC6928] and T/TCP hinted that it should be initialized to the old window size [RFC1644]. In the former cases, the assumption is that new connections should behave as conservatively as possible. In the latter T/TCP case, no accommodation is made for concurrent aggregate behavior.

In either case, the sum of window sizes can increase, rather than remain constant. A different approach is to give each pending connection its "fair share" of the available congestion window, and let the connections balance from there. The assumption we make here is that new connections are implicit requests for an equal share of available link bandwidth, which should be granted at the expense of current connections. [TBD - a new method for safe congestion sharing will be described]

8. Compatibility Issues

For the congestion and current window information, the initial values computed by TCB interdependence may not be consistent with the long-term aggregate behavior of a set of concurrent connections between the same endpoints. Under conventional TCP congestion control, if a single existing connection has converged to a congestion window of 40 segments, two newly joining concurrent

connections assume initial windows of 10 segments [RFC6928], and the current connection's window doesn't decrease to accommodate this additional load and connections can mutually interfere. One example of this is seen on low-bandwidth, high-delay links, where concurrent connections supporting Web traffic can collide because their initial windows were too large, even when set at one segment.

[TBD - this paragraph needs to be revised based on new recommendations] Under TCB interdependence, all three connections could change to use a congestion window of 12 (rounded down to an even number from 13.33, i.e., $40/3$). This would include both increasing the initial window of the new connections (vs. current recommendations [RFC6928]) and decreasing the congestion window of the current connection (from 40 down to 12). This gives the new connections a larger initial window than allowed by [RFC6928], but maintains the aggregate. Depending on whether the previous connections were in steady-state, this can result in more bursty behavior, e.g., when previous connections are idle and new connections commence with a large amount of available data to transmit. Additionally, reducing the congestion window of an existing connection needs to account for the number of packets that are already in flight.

Because this proposal attempts to anticipate the aggregate steady-state values of TCB state among a group or over time, it should avoid the transient effects of new connections. In addition, because it considers the ensemble and temporal properties of those aggregates, it should also prevent the transients of short-lived or multiple concurrent connections from adversely affecting the overall network performance. There have been ongoing analysis and experiments to validate these assumptions. For example, [Ph12] recommends to only cache ssthresh for temporal sharing when flows are long. Sharing ssthresh between short flows can deteriorate the overall performance of individual connections [Ph12, Nd16], although this may benefit overall network performance. [TBD - the details of this issue need to be summarized and clarified herein].

[TBD - placeholder for corresponding RTT discussion]

Due to mechanisms like ECMP and LAG [RFC7424], TCP connections sharing the same host-pair may not always share the same path. This does not matter for host-specific information such as RWIN and TCP option state, such as TFOinfo. When TCB information is shared across different SYN destination ports, path-related information can be incorrect; however, the impact of this error is potentially diminished if (as discussed here) TCB sharing affects only the transient event of a connection start or if TCB information is

shared only within connections to the same SYN destination port. In case of Temporal Sharing, TCB information could also become invalid over time. Because this is similar to the case when a connection becomes idle, mechanisms that address idle TCP connections (e.g., [RFC7661]) could also be applied to TCB cache management.

There may be additional considerations to the way in which TCB interdependence rebalances congestion feedback among the current connections, e.g., it may be appropriate to consider the impact of a connection being in Fast Recovery [RFC5861] or some other similar unusual feedback state, e.g., as inhibiting or affecting the calculations described herein.

TCP is sometimes used in situations where packets of the same host-pair always take the same path. Because ECMP and LAG examine TCP port numbers, they may not be supported when TCP segments are encapsulated, encrypted, or altered - for example, some Virtual Private Networks (VPNs) are known to use proprietary UDP encapsulation methods. Similarly, they cannot operate when the TCP header is encrypted, e.g., when using IPsec ESP. TCB interdependence among the entire set sharing the same endpoint IP addresses should work without problems under these circumstances. Moreover, measures to increase the probability that connections use the same path could be applied: e.g., the connections could be given the same IPv6 flow label. TCB interdependence can also be extended to sets of host IP address pairs that share the same network path conditions, such as when a group of addresses is on the same LAN (see Section 9).

It can be wrong to share TCB information between TCP connections on the same host as identified by the IP address if an IP address is assigned to a new host (e.g., IP address spinning, as is used by ISPs to inhibit running servers). It can be wrong if Network Address (and Port) Translation (NA(P)T) [RFC2663] or any other IP sharing mechanism is used. Such mechanisms are less likely to be used with IPv6. Other methods to identify a host could also be considered to make correct TCB sharing more likely. Moreover, some TCB information is about dominant path properties rather than the specific host. IP addresses may differ, yet the relevant part of the path may be the same.

9. Implications

There are several implications to incorporating TCB interdependence in TCP implementations. First, it may reduce the need for application-layer multiplexing for performance enhancement [RFC7231]. Protocols like HTTP/2 [RFC7540] avoid connection reestablishment costs by serializing or multiplexing a set of per-

host connections across a single TCP connection. This avoids TCP's per-connection OPEN handshake and also avoids recomputing MSS, RTT, and congestion windows. By avoiding the so-called, "slow-start restart," performance can be optimized. TCB interdependence can provide the "slow-start restart avoidance" of multiplexing, without requiring a multiplexing mechanism at the application layer.

TCB interdependence pushes some of the TCP implementation from the traditional transport layer (in the ISO model), to the network layer. This acknowledges that some state is in fact per-host-pair or can be per-path as indicated solely by that host-pair. Transport protocols typically manage per-application-pair associations (per stream), and network protocols manage per-host-pair and path associations (routing). Round-trip time, MSS, and congestion information could be more appropriately handled in a network-layer fashion, aggregated among concurrent connections, and shared across connection instances [RFC3124].

An earlier version of RTT sharing suggested implementing RTT state at the IP layer, rather than at the TCP layer [Ja86]. Our observations are for sharing state among TCP connections, which avoids some of the difficulties in an IP-layer solution. One such problem is determining the associated prior outgoing packet for an incoming packet, to infer RTT from the exchange. Because RTTs are still determined inside the TCP layer, this is simpler than at the IP layer. This is a case where information should be computed at the transport layer, but could be shared at the network layer.

Per-host-pair associations are not the limit of these techniques. It is possible that TCBS could be similarly shared between hosts on a subnet or within a cluster, because the predominant path can be subnet-subnet, rather than host-host. Additionally, TCB interdependence can be applied to any protocol with congestion state, including SCTP [RFC4960] and DCCP [RFC4340], as well as for individual subflows in Multipath TCP [RFC6824].

There may be other information that can be shared between concurrent connections. For example, knowing that another connection has just tried to expand its window size and failed, a connection may not attempt to do the same for some period. The idea is that existing TCP implementations infer the behavior of all competing connections, including those within the same host or subnet. One possible optimization is to make that implicit feedback explicit, via extended information associated with the endpoint IP address and its TCP implementation, rather than per-connection state in the TCB.

Like its initial version in 1997, this document's approach to TCB interdependence focuses on sharing a set of TCBs by updating the TCB state to reduce the impact of transients when connections begin or end. Other mechanisms have since been proposed to continuously share information between all ongoing communication (including connectionless protocols), updating the congestion state during any congestion-related event (e.g., timeout, loss confirmation, etc.) [RFC3124]. By dealing exclusively with transients, TCB interdependence is more likely to exhibit the same behavior as unmodified, independent TCP connections.

10. Implementation Observations

The observation that some TCB state is host-pair specific rather than application-pair dependent is not new and is a common engineering decision in layered protocol implementations. A discussion of sharing RTT information among protocols layered over IP, including UDP and TCP, occurred in [Ja86]. Although now deprecated, T/TCP was the first to propose using caches in order to maintain TCB states (see Appendix A for more information).

The table below describes the current implementation status for some TCB information in Linux kernel version 4.6, FreeBSD 10 and Windows (as of October 2016). In the table, "shared" only refers to temporal sharing.

TCB data	Status
old MMS_S	Not shared
old MMS_R	Not shared
old_sendMSS	Cached and shared in Linux (MSS)
old PMTU	Cached and shared in FreeBSD and Windows (PMTU)
old_RTT	Cached and shared in FreeBSD and Linux
old_RTTvar	Cached and shared in FreeBSD
old TFOinfo	Cached and shared in Linux and Windows
old_snd_cwnd	Not shared
old_ssthresh	Cached and shared in FreeBSD and Linux: FreeBSD: arithmetic mean of ssthresh and previous value if a previous value exists; Linux: depending on state, max(cwnd/2, ssthresh) in most cases

11. Security Considerations

These suggested implementation enhancements do not have additional ramifications for explicit attacks. These enhancements may be susceptible to denial-of-service attacks if not otherwise secured. For example, an application can open a connection and set its window size to zero, denying service to any other subsequent connection between those hosts.

TCB sharing may be susceptible to denial-of-service attacks, wherever the TCB is shared, between connections in a single host, or between hosts if TCB sharing is implemented within a subnet (see Implications section). Some shared TCB parameters are used only to create new TCBS, others are shared among the TCBS of ongoing connections. New connections can join the ongoing set, e.g., to optimize send window size among a set of connections to the same host.

Attacks on parameters used only for initialization affect only the transient performance of a TCP connection. For short connections, the performance ramification can approach that of a denial-of-

service attack. E.g., if an application changes its TCB to have a false and small window size, subsequent connections would experience performance degradation until their window grew appropriately.

The solution is to limit the effect of compromised TCB values. TCBs are compromised when they are modified directly by an application or transmitted between hosts via unauthenticated means (e.g., by using a dirty flag). TCBs that are not compromised by application modification do not have any unique security ramifications. Note that the proposed parameters for TCB sharing are not currently modifiable by an application.

All shared TCBs MUST be validated against default minimum parameters before used for new connections. This validation would not impact performance, because it occurs only at TCB initialization. This limits the effect of attacks on new connections to reducing the benefit of TCB sharing, resulting in the current default TCP performance. For ongoing connections, the effect of incoming packets on shared information should be both limited and validated against constraints before use. This is a beneficial precaution for existing TCP implementations as well.

TCBs modified by an application SHOULD NOT be shared, unless the new connection sharing the compromised information has been given explicit permission to use such information by the connection API. No mechanism for that indication currently exists, but it could be supported by an augmented API. This sharing restriction SHOULD be implemented in both the host and the subnet. Sharing on a subnet SHOULD utilize authentication to prevent undetected tampering of shared TCB parameters. These restrictions limit the security impact of modified TCBs both for connection initialization and for ongoing connections.

Finally, shared values MUST be limited to performance factors only. Other information, such as TCP sequence numbers, when shared, are already known to compromise security.

12. IANA Considerations

There are no IANA implications or requests in this document.

This section should be removed upon final publication as an RFC.

13. References

13.1. Normative References

- [RFC793] Postel, Jon, "Transmission Control Protocol," Network Working Group RFC-793/STD-7, ISI, Sept. 1981.
- [RFC1191] Mogul, J., Deering, S., "Path MTU Discovery," RFC 1191, Nov. 1990.
- [RFC1981] McCann, J., Deering, S., Mogul, J., "Path MTU Discovery for IP version 6," RFC 1981, Aug. 1996.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC4821] Mathis, M., Heffner, J., "Packetization Layer Path MTU Discovery," RFC 4821, Mar. 2007.
- [RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., Jain, A., "TCP Fast Open", RFC 7413, Dec. 2014.

13.2. Informative References

- [Br02] Brownlee, N. and K. Claffy, "Understanding Internet Traffic Streams: Dragonflies and Tortoises", IEEE Communications Magazine p110-117, 2002.
- [Be94] Berners-Lee, T., et al., "The World-Wide Web," Communications of the ACM, V37, Aug. 1994, pp. 76-82.
- [Br94] Braden, B., "T/TCP -- Transaction TCP: Source Changes for Sun OS 4.1.3," Release 1.0, USC/ISI, September 14, 1994.
- [Co91] Comer, D., Stevens, D., Internetworking with TCP/IP, V2, Prentice-Hall, NJ, 1991.
- [FreeBSD] FreeBSD source code, Release 2.10, <http://www.freebsd.org/>
- [Ja86] Jacobson, V., (mail to public list "tcp-ip", no archive found), 1986.
- [Nd16] Dukkipati, N., Yuchung C., and Amin V., "Research Impacting the Practice of Congestion Control." ACM SIGCOMM CCR (editorial).

- [DM16] Matz, D., "Optimize TCP's Minimum Retransmission Timeout for Low Latency Environments", Master's thesis, Technical University Munich, 2016.
- [Ph12] Hurtig, P., Brunstrom, A., "Enhanced metric caching for short TCP flows," 2012 IEEE International Conference on Communications (ICC), Ottawa, ON, 2012, pp. 1209-1213.
- [RFC1122] Braden, R. (ed), "Requirements for Internet Hosts -- Communication Layers", RFC-1122, Oct. 1989.
- [RFC1644] Braden, R., "T/TCP -- TCP Extensions for Transactions Functional Specification," RFC-1644, July 1994.
- [RFC1379] Braden, R., "Transaction TCP -- Concepts," RFC-1379, September 1992.
- [RFC2663] Srisuresh, P., Holdrege, M., "IP Network Address Translator (NAT) Terminology and Considerations", RFC-2663, August 1999.
- [RFC3390] Allman, M., Floyd, S., Partridge, C., "Increasing TCP's Initial Window," RFC 3390, Oct. 2002.
- [RFC7231] Fielding, R., J. Reshke, Eds., "HTTP/1.1 Semantics and Content," RFC-7231, June 2014.
- [RFC3124] Balakrishnan, H., Seshan, S., "The Congestion Manager," RFC 3124, June 2001.
- [RFC4340] Kohler, E., Handley, M., Floyd, S., "Datagram Congestion Control Protocol (DCCP)," RFC 4340, Mar. 2006.
- [RFC4960] Stewart, R., (Ed.), "Stream Control Transmission Protocol," RFC4960, Sept. 2007.
- [RFC5861] Allman, M., Paxson, V., Blanton, E., "TCP Congestion Control," RFC 5861, Sept. 2009.
- [RFC5925] Touch, J., Mankin, A., Bonica, R., "The TCP Authentication Option," RFC 5925, June 2010.
- [RFC6824] Ford, A., Raiciu, C., Handley, M., Bonaventure, O., "TCP Extensions for Multipath Operation with Multiple Addresses," RFC 6824, Jan. 2013.

- [RFC6928] Chu, J., Dukkupati, N., Cheng, Y., Mathis, M., "Increasing TCP's Initial Window," RFC 6928, Apr. 2013.
- [RFC7424] Krishnan, R., Yong, L., Ghanwani, A., So, N., Khasnabish, B., "Mechanisms for Optimizing Link Aggregation Group (LAG) and Equal-Cost Multipath (ECMP) Component Link Utilization in Networks", RFC 7424, Jan. 2015
- [RFC7540] Belshe, M., Peon, R., Thomson, M., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, May 2015.
- [RFC7661] Fairhurst, G., Sathiaselan, A., Secchi, R., "Updating TCP to Support Rate-Limited Traffic", RFC 7661, Oct. 2015

14. Acknowledgments

The authors would like to thank for Praveen Balasubramanian for information regarding TCB sharing in Windows, and Yuchung Cheng, Lars Eggert, Ilpo Jarvinen and Michael Scharf for comments on earlier versions of the draft. This work has received funding from a collaborative research project between the University of Oslo and Huawei Technologies Co., Ltd., and is partly supported by USC/ISI's Postel Center.

This document was prepared using 2-Word-v2.0.template.dot.

15. Change log

from -01 to -02:

- Stated that our OS implementation overview table only covers temporal sharing.
- Correctly reflected sharing of old_RTT in Linux in the implementation overview table.
- Marked entries that are considered safe to share with an asterisk (suggestion was to split the table)
- Discussed correct host identification: NATs may make IP addresses the wrong input, could e.g. use HTTP cookie.
- Included MMS_S and MMS_R from RFC1122; fixed the use of MSS and MTU

- Added information about option sharing, listed options in the appendix

Authors' Addresses

Joe Touch
USC/ISI
4676 Admiralty Way
Marina del Rey, CA 90292-6695
USA

Phone: +1 (310) 448-9151
Email: touch@isi.edu

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

Jianjie You
Huawei
101 Software Avenue, Yuhua District
Nanjing 210012
China

Email: youjianjie@huawei.com

16. Appendix A: TCB sharing history

T/TCP proposed using caches to maintain TCB information across instances (temporal sharing), e.g., smoothed RTT, RTT variance, congestion avoidance threshold, and MSS [RFC1644]. These values were in addition to connection counts used by T/TCP to accelerate data delivery prior to the full three-way handshake during an OPEN. The goal was to aggregate TCB components where they reflect one association - that of the host-pair, rather than artificially separating those components by connection.

At least one T/TCP implementation saved the MSS and aggregated the RTT parameters across multiple connections, but omitted caching the congestion window information [Br94], as originally specified in [RFC1379]. Some T/TCP implementations immediately updated MSS when the TCP MSS header option was received [Br94], although this was not addressed specifically in the concepts or functional specification [RFC1379][RFC1644]. In later T/TCP implementations, RTT values were updated only after a CLOSE, which does not benefit concurrent sessions.

Temporal sharing of cached TCB data was originally implemented in the SunOS 4.1.3 T/TCP extensions [Br94] and the FreeBSD port of same [FreeBSD]. As mentioned before, only the MSS and RTT parameters were cached, as originally specified in [RFC1379]. Later discussion of T/TCP suggested including congestion control parameters in this cache [RFC1644].

17. Appendix B: Options

In addition to the options that can be cached and shared, this memo also lists all options for which state should **not** be kept. This list is meant to avoid work duplication and should be removed upon publication.

Obsolete (MUST NOT keep state):

ECHO

ECHO REPLY

PO Conn permitted

PO service profile

CC

CC.NEW

CC.ECHO

Alt CS req

Alt CS data

No state to keep:

EOL

NOP

WS

SACK

TS

MD5

TCP-AO

EXP1

EXP2

MUST NOT keep state:

Skeeter (DH exchange - might be obsolete, though)

Bubba (DH exchange - might really be obsolete, though)

Trailer CS

SCPS capabilities

S-NACK

Records boundaries

Corruption experienced

SNAP

TCP Compression

Quickstart response

UTO

MPTCP (can we cache when this fails?)

TFO success

MAY keep state:

MSS

TFO failure (so we don't try again, since it's optional)

MUST keep state:

TFP cookie (if TFO succeeded in the past)

