

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

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

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

Abstract

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

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 March 18, 2019.

Copyright Notice

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

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

Table of Contents

1. Introduction	2
2. Definitions	3
3. Specification	4
3.1. Choice of ABE Multiplier	4
4. Discussion	6
4.1. Why Use ECN to Vary the Degree of Backoff?	6
4.2. An RTT-based response to indicated congestion	7
5. ABE Deployment Requirements	7
6. ABE Experiment Goals	8
7. Acknowledgements	8
8. IANA Considerations	9
9. Implementation Status	9
10. Security Considerations	9
11. Revision Information	9
12. References	11
12.1. Normative References	11
12.2. Informative References	11
Authors' Addresses	13

1. Introduction

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

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

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

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

2. Definitions

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

3. Specification

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

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

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

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

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

3.1. Choice of ABE Multiplier

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

In other words, in response to inferred packet loss:

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

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

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

and

```
  cwnd = ssthresh
```

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

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

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

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

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

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

4. Discussion

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

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

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

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

The idea of reacting differently to inferred packet loss and detection of an ECN-signalled congestion pre-dates this specification. For example, previous research proposed using ECN CE-marked feedback to modify TCP congestion control behaviour via a larger multiplicative decrease factor in conjunction with a smaller additive increase factor [ICC2002]. The goal of this former work was to operate across AQM bottlenecks using Random Early Detection (RED)

that were not necessarily configured to emulate a short queue (The current usage of RED as an Internet AQM method is limited [RFC7567]).

4.2. An RTT-based response to indicated congestion

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

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

5. ABE Deployment Requirements

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

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

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

6. ABE Experiment Goals

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

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

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

7. Acknowledgements

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

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

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

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

8. IANA Considerations

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

This document includes no request to IANA.

9. Implementation Status

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

10. Security Considerations

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

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

11. Revision Information

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

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

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

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

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

-07. Addressed comments following WGLC.

- o Updated Reference citations.
 - o Removed paragraph containing a wrong statement related to timeout in section 4.1.
 - o Discuss what happens when $cwnd \leq ssthresh$.
 - o Added text on Concern about lower bound of $2 \cdot SMSS$.
- 06. Addressed Michael Scharf's comments.
- 05. Refined the description of the experiment based on feedback at IETF-100. Incorporated comments from David Black.
- 04. Incorporates review comments from Lawrence Stewart and the remaining comments from Roland Bless. References are updated.
- 03. Several review comments from Roland Bless are addressed. Consistent terminology and equations. Clarification on the scope of recommended $\beta_{\{ecn\}}$ value.
- 02. Corrected the equations in Section 3.1. Updated the affiliations. Lower bound for $cwnd$ is defined. A recommendation for window-based transport protocols is changed to cover all transport protocols that implement a congestion control reduction to an ECN congestion signal. Added text about ABE's FreeBSD mainline kernel status including a reference to the FreeBSD code review page. References are updated.
- 01. Text improved, mainly incorporating comments from Stuart Cheshire. The reference to a technical report has been updated to a published version of the tests [ABE2017]. Used "AQM Mechanism" throughout in place of other alternatives, and more consistent use of technical language and clarification on the intended purpose of the experiments required by EXP status. There was no change to the technical content.
- 00. draft-ietf-tcpm-alternativebackoff-ecn-00 replaces draft-khademi-tcpm-alternativebackoff-ecn-01. Text describing the nature of the experiment was added.

Individual draft -01. This I-D now refers to draft-black-tsvwg-ecn-experimentation-02, which replaces draft-khademi-tsvwg-ecn-response-00 to make a broader update to RFC 3168 for the sake of allowing experiments. As a result, some of the motivating and discussing text that was moved from draft-khademi-alternativebackoff-ecn-03 to draft-khademi-tsvwg-ecn-response-00 has now been re-inserted here.

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

12. References

12.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <<https://www.rfc-editor.org/info/rfc3168>>.
- [RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <<https://www.rfc-editor.org/info/rfc5681>>.
- [RFC7567] Baker, F., Ed. and G. Fairhurst, Ed., "IETF Recommendations Regarding Active Queue Management", BCP 197, RFC 7567, DOI 10.17487/RFC7567, July 2015, <<https://www.rfc-editor.org/info/rfc7567>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8257] Bensley, S., Thaler, D., Balasubramanian, P., Eggert, L., and G. Judd, "Data Center TCP (DCTCP): TCP Congestion Control for Data Centers", RFC 8257, DOI 10.17487/RFC8257, October 2017, <<https://www.rfc-editor.org/info/rfc8257>>.
- [RFC8311] Black, D., "Relaxing Restrictions on Explicit Congestion Notification (ECN) Experimentation", RFC 8311, DOI 10.17487/RFC8311, January 2018, <<https://www.rfc-editor.org/info/rfc8311>>.

12.2. Informative References

- [ABE-FreeBSD] "ABE patch review in FreeBSD", <<https://svnweb.freebsd.org/base?view=revision&revision=331214>>.

- [ABE2017] Khademi, N., Armitage, G., Welzl, M., Fairhurst, G., Zander, S., and D. Ros, "Alternative Backoff: Achieving Low Latency and High Throughput with ECN and AQM", IFIP NETWORKING 2017, Stockholm, Sweden, June 2017.
- [BUFFERBLOAT] Gettys, J. and K. Nichols, "Bufferbloat: Dark Buffers in the Internet", ACM Queue 9, 11, DOI 10.1145/2063166.2071893; <https://queue.acm.org/detail.cfm?id=2071893>, November 2011.
- [CODEL2012] Nichols, K. and V. Jacobson, "Controlling Queue Delay", July 2012, <<http://queue.acm.org/detail.cfm?id=2209336>>.
- [I-D.ietf-tcpm-accurate-ecn] Briscoe, B., Kuehlewind, M., and R. Scheffenegger, "More Accurate ECN Feedback in TCP", draft-ietf-tcpm-accurate-ecn-06 (work in progress), March 2018.
- [ICC2002] Kwon, M. and S. Fahmy, "TCP Increase/Decrease Behavior with Explicit Congestion Notification (ECN)", IEEE ICC 2002, New York, New York, USA, May 2002, <<http://dx.doi.org/10.1109/ICC.2002.997262>>.
- [RFC8033] Pan, R., Natarajan, P., Baker, F., and G. White, "Proportional Integral Controller Enhanced (PIE): A Lightweight Control Scheme to Address the Bufferbloat Problem", RFC 8033, DOI 10.17487/RFC8033, February 2017, <<https://www.rfc-editor.org/info/rfc8033>>.
- [RFC8087] Fairhurst, G. and M. Welzl, "The Benefits of Using Explicit Congestion Notification (ECN)", RFC 8087, DOI 10.17487/RFC8087, March 2017, <<https://www.rfc-editor.org/info/rfc8087>>.
- [RFC8289] Nichols, K., Jacobson, V., McGregor, A., Ed., and J. Iyengar, Ed., "Controlled Delay Active Queue Management", RFC 8289, DOI 10.17487/RFC8289, January 2018, <<https://www.rfc-editor.org/info/rfc8289>>.
- [RFC8312] Rhee, I., Xu, L., Ha, S., Zimmermann, A., Eggert, L., and R. Scheffenegger, "CUBIC for Fast Long-Distance Networks", RFC 8312, DOI 10.17487/RFC8312, February 2018, <<https://www.rfc-editor.org/info/rfc8312>>.

Authors' Addresses

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

Email: naeemk@ifi.uio.no

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

Email: michawe@ifi.uio.no

Grenville Armitage
Netflix Inc.

Email: garmitage@netflix.com

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

Email: gorry@erg.abdn.ac.uk

TCP Maintenance Working Group
Internet-Draft
Intended status: Standards Track
Expires: June 25, 2021

Y. Cheng
N. Cardwell
N. Dukkipati
P. Jha
Google, Inc
December 22, 2020

The RACK-TLP loss detection algorithm for TCP
draft-ietf-tcpm-rack-15

Abstract

This document presents the RACK-TLP loss detection algorithm for TCP. RACK-TLP uses per-segment transmit timestamps and selective acknowledgements (SACK) and has two parts: RACK ("Recent ACKnowledgment") starts fast recovery quickly using time-based inferences derived from ACK feedback. TLP ("Tail Loss Probe") leverages RACK and sends a probe packet to trigger ACK feedback to avoid retransmission timeout (RTO) events. Compared to the widely used DUPACK threshold approach, RACK-TLP detects losses more efficiently when there are application-limited flights of data, lost retransmissions, or data packet reordering events. It is intended to be an alternative to the DUPACK threshold approach.

Status of This Memo

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 June 25, 2021.

Copyright Notice

Copyright (c) 2020 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. Terminology	3
2. Introduction	3
2.1. Background	3
2.2. Motivation	4
3. RACK-TLP high-level design	5
3.1. RACK: time-based loss inferences from ACKs	5
3.2. TLP: sending one segment to probe losses quickly with RACK	6
3.3. RACK-TLP: reordering resilience with a time threshold	6
3.3.1. Reordering design rationale	6
3.3.2. Reordering window adaptation	8
3.4. An Example of RACK-TLP in Action: fast recovery	9
3.5. An Example of RACK-TLP in Action: RTO	10
3.6. Design Summary	10
4. Requirements	11
5. Definitions	11
5.1. Terms	11
5.2. Per-segment variables	12
5.3. Per-connection variables	12
5.4. Per-connection timers	13
6. RACK Algorithm Details	13
6.1. Upon transmitting a data segment	13
6.2. Upon receiving an ACK	14
6.3. Upon RTO expiration	20
7. TLP Algorithm Details	21
7.1. Initializing state	21
7.2. Scheduling a loss probe	21
7.3. Sending a loss probe upon PTO expiration	22
7.4. Detecting losses using the ACK of the loss probe	24
7.4.1. General case: detecting packet losses using RACK	24
7.4.2. Special case: detecting a single loss repaired by the loss probe	24
8. Managing RACK-TLP timers	25
9. Discussion	25
9.1. Advantages and disadvantages	25
9.2. Relationships with other loss recovery algorithms	27

9.3. Interaction with congestion control	28
9.4. TLP recovery detection with delayed ACKs	29
9.5. RACK-TLP for other transport protocols	29
10. Security Considerations	30
11. IANA Considerations	30
12. Acknowledgments	30
13. References	30
13.1. Normative References	30
13.2. Informative References	31
Authors' Addresses	32

1. Terminology

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

2. Introduction

This document presents RACK-TLP, a TCP loss detection algorithm that improves upon the widely implemented duplicate acknowledgment (DUPACK) counting approach in [RFC5681] [RFC6675], and that is RECOMMENDED to be used as an alternative to that earlier approach. RACK-TLP has two parts: RACK ("Recent ACKnowledgment") detects losses quickly using time-based inferences derived from ACK feedback. TLP ("Tail Loss Probe") triggers ACK feedback by quickly sending a probe segment, to avoid retransmission timeout (RTO) events.

2.1. Background

In traditional TCP loss recovery algorithms [RFC5681] [RFC6675], a sender starts fast recovery when the number of DUPACKs received reaches a threshold (DupThresh) that defaults to 3 (this approach is referred to as DUPACK-counting in the rest of the document). The sender also halves the congestion window during the recovery. The rationale behind the partial window reduction is that congestion does not seem severe since ACK clocking is still maintained. The time elapsed in fast recovery can be just one round-trip, e.g. if the sender uses SACK-based recovery [RFC6675] and the number of lost segments is small.

If fast recovery is not triggered, or triggers but fails to repair all the losses, then the sender resorts to RTO recovery. The RTO timer interval is conservatively the smoothed RTT (SRTT) plus four times the RTT variation, and is lower bounded to 1 second [RFC6298]. Upon RTO timer expiration, the sender retransmits the first

unacknowledged segment and resets the congestion window to the LOSS WINDOW value (by default 1 full-size segment [RFC5681]). The rationale behind the congestion window reset is that an entire flight of data was lost, and the ACK clock was lost, so this deserves a cautious response. The sender then retransmits the rest of the data following the slow start algorithm [RFC5681]. The time elapsed in RTO recovery is one RTO interval plus the number of round-trips needed to repair all the losses.

2.2. Motivation

Fast Recovery is the preferred form of loss recovery because it can potentially recover all losses in the time scale of a single round trip, with only a fractional congestion window reduction. RTO recovery and congestion window reset should ideally be the last resort, only used when the entire flight is lost. However, in addition to losing an entire flight of data, the following situations can unnecessarily resort to RTO recovery with traditional TCP loss recovery algorithms [RFC5681] [RFC6675]:

1. Packet drops for short flows or at the end of an application data flight. When the sender is limited by the application (e.g. structured request/response traffic), segments lost at the end of the application data transfer often can only be recovered by RTO. Consider an example of losing only the last segment in a flight of 100 segments. Lacking any DUPACK, the sender RTO expires and reduces the congestion window to 1, and raises the congestion window to just 2 after the loss repair is acknowledged. In contrast, any single segment loss occurring between the first and the 97th segment would result in fast recovery, which would only cut the window in half.
2. Lost retransmissions. Heavy congestion or traffic policers can cause retransmissions to be lost. Lost retransmissions cause a resort to RTO recovery, since DUPACK-counting does not detect the loss of the retransmissions. Then the slow start after RTO recovery could cause burst losses again that severely degrades performance [POLICER16].
3. Packet reordering. In this document, "reordering" refers to the events where segments are delivered at the TCP receiver in a chronological order different from their chronological transmission order. Link-layer protocols (e.g., 802.11 block ACK), link bonding, or routers' internal load-balancing (e.g., ECMP) can deliver TCP segments out of order. The degree of such reordering is usually within the order of the path round trip time. If the reordering degree is beyond DupThresh, DUPACK-counting can cause a spurious fast recovery and unnecessary

congestion window reduction. To mitigate the issue, TCP-NCR [RFC4653] increases the DupThresh from the current fixed value of three duplicate ACKs [RFC5681] to approximately a congestion window of data having left the network.

3. RACK-TLP high-level design

RACK-TLP allows senders to recover losses more effectively in all three scenarios described in the previous section. There are two design principles behind RACK-TLP. The first principle is to detect losses via ACK events as much as possible, to repair losses at round-trip time-scales. The second principle is to gently probe the network to solicit additional ACK feedback, to avoid RTO expiration and subsequent congestion window reset. At a high level, the two principles are implemented in RACK and TLP, respectively.

3.1. RACK: time-based loss inferences from ACKs

The rationale behind RACK is that if a segment is delivered out of order, then the segments sent chronologically before that were either lost or reordered. This concept is not fundamentally different from [RFC5681] [RFC6675] [FACK]. RACK's key innovation is using per-segment transmission timestamps and widely-deployed SACK [RFC2018] options to conduct time-based inferences, instead of inferring losses by counting ACKs or SACKed sequences. Time-based inferences are more robust than DUPACK-counting approaches because they have no dependence on flight size, and thus are effective for application-limited traffic.

Conceptually, RACK keeps a virtual timer for every data segment sent (including retransmissions). Each timer expires dynamically based on the latest RTT measurements plus an additional delay budget to accommodate potential packet reordering (called the reordering window). When a segment's timer expires, RACK marks the corresponding segment lost for retransmission.

In reality, as an algorithm, RACK does not arm a timer for every segment sent because it's not necessary. Instead the sender records the most recent transmission time of every data segment sent, including retransmissions. For each ACK received, the sender calculates the latest RTT measurement (if eligible) and adjusts the expiration time of every segment sent but not yet delivered. If a segment has expired, RACK marks it lost.

Since the time-based logic of RACK applies equally to retransmissions and original transmissions, it can detect lost retransmissions as well. If a segment has been retransmitted but its most recent

(re)transmission timestamp has expired, then after a reordering window it's marked lost.

3.2. TLP: sending one segment to probe losses quickly with RACK

RACK infers losses from ACK feedback; however, in some cases ACKs are sparse, particularly when the inflight is small or when the losses are high. In some challenging cases the last few segments in a flight are lost. With [RFC5681] or [RFC6675] the sender's RTO would expire and reset the congestion window, when in reality most of the flight has been delivered.

Consider an example where a sender with a large congestion window transmits 100 new data segments after an application write, and only the last three segments are lost. Without RACK-TLP, the RTO expires, the sender retransmits the first unacknowledged segment, and the congestion window slow-starts from 1. After all the retransmits are acknowledged the congestion window has been increased to 4. The total delivery time for this application transfer is three RTTs plus one RTO, a steep cost given that only a tiny fraction of the flight was lost. If instead the losses had occurred three segments sooner in the flight, then fast recovery would have recovered all losses within one round-trip and would have avoided resetting the congestion window.

Fast Recovery would be preferable in such scenarios; TLP is designed to trigger the feedback RACK needed to enable that. After the last (100th) segment was originally sent, TLP sends the next available (new) segment or retransmits the last (highest-sequenced) segment in two round-trips to probe the network, hence the name "Tail Loss Probe". The successful delivery of the probe would solicit an ACK. RACK uses this ACK to detect that the 98th and 99th segments were lost, trigger fast recovery, and retransmit both successfully. The total recovery time is four RTTs, and the congestion window is only partially reduced instead of being fully reset. If the probe was also lost then the sender would invoke RTO recovery resetting the congestion window.

3.3. RACK-TLP: reordering resilience with a time threshold

3.3.1. Reordering design rationale

Upon receiving an ACK indicating an out-of-order data delivery, a sender cannot tell immediately whether that out-of-order delivery was a result of reordering or loss. It can only distinguish between the two in hindsight if the missing sequence ranges are filled in later without retransmission. Thus a loss detection algorithm needs to

budget some wait time -- a reordering window -- to try to disambiguate packet reordering from packet loss.

The reordering window in the DUPACK-counting approach is implicitly defined as the elapsed time to receive acknowledgements for DupThresh-worth of out-of-order deliveries. This approach is effective if the network reordering degree (in sequence distance) is smaller than DupThresh and at least DupThresh segments after the loss are acknowledged. For cases where the reordering degree is larger than the default DupThresh of 3 packets, one alternative is to dynamically adapt DupThresh based on the FlightSize (e.g., the sender adjusts the DUPTHRESH to half of the FlightSize). However, this does not work well with the following two types of reordering:

1. Application-limited flights where the last non-full-sized segment is delivered first and then the remaining full-sized segments in the flight are delivered in order. This reordering pattern can occur when segments traverse parallel forwarding paths. In such scenarios the degree of reordering in packet distance is one segment less than the flight size.
2. A flight of segments that are delivered partially out of order. One cause for this pattern is wireless link-layer retransmissions with an inadequate reordering buffer at the receiver. In such scenarios, the wireless sender sends the data packets in order initially, but some are lost and then recovered by link-layer retransmissions; the wireless receiver delivers the TCP data packets in the order they are received, due to the inadequate reordering buffer. The random wireless transmission errors in such scenarios cause the reordering degree, expressed in packet distance, to have highly variable values up to the flight size.

In the above two cases the degree of reordering in packet distance is highly variable. This makes the DUPACK-counting approach ineffective including dynamic adaptation variants like [RFC4653]. Instead the degree of reordering in time difference in such cases is usually within a single round-trip time. This is because the packets either traverse slightly disjoint paths with similar propagation delays or are repaired quickly by the local access technology. Hence, using a time threshold instead of packet threshold strikes a middle ground, allowing a bounded degree of reordering resilience while still allowing fast recovery. This is the rationale behind the RACK-TLP reordering resilience design.

Specifically, RACK-TLP introduces a new dynamic reordering window parameter in time units, and the sender considers a data segment *S* lost if both conditions are met:

1. Another data segment sent later than S has been delivered
2. S has not been delivered after the estimated round-trip time plus the reordering window

Note that condition (1) implies at least one round-trip of time has elapsed since S has been sent.

3.3.2. Reordering window adaptation

The RACK reordering window adapts to the measured duration of reordering events, within reasonable and specific bounds to disincentivize excessive reordering. More specifically, the sender sets the reordering window as follows:

1. The reordering window SHOULD be set to zero if no reordering has been observed on the connection so far, and either (a) three segments have been delivered out of order since the last recovery or (b) the sender is already in fast or RTO recovery. Otherwise, the reordering window SHOULD start from a small fraction of the round trip time, or zero if no round trip time estimate is available.
2. The RACK reordering window SHOULD adaptively increase (using the algorithm in "Step 4: Update RACK reordering window", below) if the sender receives a Duplicate Selective Acknowledgement (DSACK) option [RFC2883]. Receiving a DSACK suggests the sender made a spurious retransmission, which may have been due to the reordering window being too small.
3. The RACK reordering window MUST be bounded and this bound SHOULD be SRTT.

Rules 2 and 3 are required to adapt to reordering caused by dynamics such as the prolonged link-layer loss recovery episodes described earlier. Each increase in the reordering window requires a new round trip where the sender receives a DSACK; thus, depending on the extent of reordering, it may take multiple round trips to fully adapt.

For short flows, the low initial reordering window helps recover losses quickly, at the risk of spurious retransmissions. The rationale is that spurious retransmissions for short flows are not expected to produce excessive additional network traffic. For long flows the design tolerates reordering within a round trip. This handles reordering in small time scales (reordering within the round-trip time of the shortest path).

However, the fact that the initial reordering window is low, and the reordering window's adaptive growth is bounded, means that there will continue to be a cost to reordering that disincentivizes excessive reordering.

3.4. An Example of RACK-TLP in Action: fast recovery

The following example in figure 1 illustrates the RACK-TLP algorithm in action:

Event	TCP DATA SENDER	TCP DATA RECEIVER
1.	Send P0, P1, P2, P3 [P1, P2, P3 dropped by network]	-->
2.		<-- Receive P0, ACK P0
3a.	2RTTs after (2), TLP timer fires	
3b.	TLP: retransmits P3	-->
4.		<-- Receive P3, SACK P3
5a.	Receive SACK for P3	
5b.	RACK: marks P1, P2 lost	
5c.	Retransmit P1, P2 [P1 retransmission dropped by network]	-->
6.		<-- Receive P2, SACK P2 & P3
7a.	RACK: marks P1 retransmission lost	
7b.	Retransmit P1	-->
8.		<-- Receive P1, ACK P3

Figure 1. RACK-TLP protocol example

Figure 1, above, illustrates a sender sending four segments (P1, P2, P3, P4) and losing the last three segments. After two round-trips, TLP sends a loss probe, retransmitting the last segment, P3, to solicit SACK feedback and restore the ACK clock (event 3). The delivery of P3 enables RACK to infer (event 5b) that P1 and P2 were likely lost, because they were sent before P3. The sender then retransmits P1 and P2. Unfortunately, the retransmission of P1 is lost again. However, the delivery of the retransmission of P2 allows RACK to infer that the retransmission of P1 was likely lost (event 7a), and hence P1 should be retransmitted (event 7b). Note that [RFC5681] mandates a principle that loss in two successive windows of data, or the loss of a retransmission, must be taken as two

indications of congestion and therefore result in two separate congestion control reactions.

3.5. An Example of RACK-TLP in Action: RTO

In addition to enhancing fast recovery, RACK improves the accuracy of RTO recovery by reducing spurious retransmissions.

Without RACK, upon RTO timer expiration the sender marks all the unacknowledged segments lost. This approach can lead to spurious retransmissions. For example, consider a simple case where one segment was sent with an RTO of 1 second, and then the application writes more data, causing a second and third segment to be sent right before the RTO of the first segment expires. Suppose none of the segments were lost. Without RACK, if there is a spurious RTO then the sender marks all three segments as lost and retransmits the first segment. If the ACK for the original copy of the first segment arrives right after the spurious RTO retransmission, then the sender continues slow start and spuriously retransmits the second and third segments, since it (erroneously) presumed they are lost.

With RACK, upon RTO timer expiration the only segment automatically marked lost is the first segment (since it was sent an RTO ago); for all the other segments RACK only marks the segment lost if at least one round trip has elapsed since the segment was transmitted. Consider the previous example scenario, this time with RACK. With RACK, when the RTO expires the sender only marks the first segment as lost, and retransmits that segment. The other two very recently sent segments are not marked lost, because they were sent less than one round trip ago and there were no ACKs providing evidence that they were lost. Upon receiving the ACK for the RTO retransmission the RACK sender would not yet retransmit the second or third segment. but rather would rearm the RTO timer and wait for a new RTO interval to elapse before marking the second or third segments as lost.

3.6. Design Summary

To summarize, RACK-TLP aims to adapt to small time-varying degrees of reordering, quickly recover most losses within one to two round trips, and avoid costly RTO recoveries. In the presence of reordering, the adaptation algorithm can impose sometimes-needless delays when it waits to disambiguate loss from reordering, but the penalty for waiting is bounded to one round trip and such delays are confined to flows long enough to have observed reordering.

4. Requirements

The reader is expected to be familiar with the definitions given in the TCP congestion control [RFC5681] and selective acknowledgment [RFC2018] and loss recovery [RFC6675] RFCs. RACK-TLP has the following requirements:

1. The connection MUST use selective acknowledgment (SACK) options [RFC2018], and the sender MUST keep SACK scoreboard information on a per-connection basis ("SACK scoreboard" has the same meaning here as in [RFC6675] section 3).
2. For each data segment sent, the sender MUST store its most recent transmission time with a timestamp whose granularity is finer than 1/4 of the minimum RTT of the connection. At the time of writing, microsecond resolution is suitable for intra-datacenter traffic and millisecond granularity or finer is suitable for the Internet. Note that RACK-TLP can be implemented with TSO (TCP Segmentation Offload) support by having multiple segments in a TSO aggregate share the same timestamp.
3. RACK DSACK-based reordering window adaptation is RECOMMENDED but is not required.
4. TLP requires RACK.

5. Definitions

The reader is expected to be familiar with the variables SND.UNA, SND.NXT, SEG.ACK, and SEG.SEQ in [RFC793], SMSS, FlightSize in [RFC5681], DupThresh in [RFC6675], RTO and SRTT in [RFC6298]. A RACK-TLP implementation uses several new terms and needs to store new per-segment and per-connection state, described below.

5.1. Terms

These terms are used to explain variables and algorithms below:

"RACK.segment". Among all the segments that have been either selectively or cumulatively acknowledged, the term RACK.segment denotes the segment that was sent most recently (including retransmissions).

"RACK.ack_ts" denotes the time when the full sequence range of RACK.segment was selectively or cumulatively acknowledged.

5.2. Per-segment variables

These variables indicate the status of the most recent transmission of a data segment:

"Segment.lost" is true if the most recent (re)transmission of the segment has been marked lost and needs to be retransmitted. False otherwise.

"Segment.retransmitted" is true if the segment has ever been retransmitted. False otherwise.

"Segment.xmit_ts" is the time of the last transmission of a data segment, including retransmissions, if any, with a clock granularity specified in the Requirements section. A maximum value INFINITE_TS indicates an invalid timestamp that represents that the Segment is not currently in flight.

"Segment.end_seq" is the next sequence number after the last sequence number of the data segment.

5.3. Per-connection variables

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

"RACK.end_seq" is the Segment.end_seq of RACK.segment.

"RACK.segs_sacked" returns the total number of segments selectively acknowledged in the SACK scoreboard.

"RACK.fack" is the highest selectively or cumulatively acknowledged sequence (i.e. forward acknowledgement).

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

"RACK.rtt" is the RTT of the most recently delivered segment on the connection (either cumulatively acknowledged or selectively acknowledged) that was not marked invalid as a possible spurious retransmission.

"RACK.reordering_seen" indicates whether the sender has detected data segment reordering event(s).

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

"RACK.dsack_round" indicates if a DSACK option has been received in the latest round trip.

"RACK.reo_wnd_mult" is the multiplier applied to adjust RACK.reo_wnd.

"RACK.reo_wnd_persist" is the number of loss recoveries before resetting RACK.reo_wnd.

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

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

"TLP.max_ack_delay": sender's maximum delayed ACK timer budget.

5.4. Per-connection timers

"RACK reordering timer": a timer that allows RACK to wait for reordering to resolve, to try to disambiguate reordering from loss, when some out-of-order segments are marked as SACKed.

"TLP PTO": a timer event indicating that an ACK is overdue and the sender should transmit a TLP segment, to solicit SACK or ACK feedback.

These timers augment the existing timers maintained by a sender, including the RTO timer [RFC6298]. A RACK-TLP sender arms one of these three timers -- RACK reordering timer, TLP PTO timer, or RTO timer -- when it has unacknowledged segments in flight. The implementation can simplify managing all three timers by multiplexing a single timer among them with an additional variable to indicate the event to invoke upon the next timer expiration.

6. RACK Algorithm Details

6.1. Upon transmitting a data segment

Upon transmitting a new segment or retransmitting an old segment, record the time in Segment.xmit_ts and set Segment.lost to FALSE. Upon retransmitting a segment, set Segment.retransmitted to TRUE.

```
RACK_transmit_new_data(Segment):  
    Segment.xmit_ts = Now()  
    Segment.lost = FALSE  
  
RACK_retransmit_data(Segment):  
    Segment.retransmitted = TRUE  
    Segment.xmit_ts = Now()  
    Segment.lost = FALSE
```

6.2. Upon receiving an ACK

Step 1: Update RACK.min_RTT.

Use the RTT measurements obtained via [RFC6298] or [RFC7323] to update the estimated minimum RTT in RACK.min_RTT. The sender SHOULD track a windowed min-filtered estimate of recent RTT measurements that can adapt when migrating to significantly longer paths, rather than a simple global minimum of all RTT measurements.

Step 2: Update state for most recently sent segment that has been delivered

In this step, RACK updates the states that track the most recently sent segment that has been delivered: RACK.segment; RACK maintains its latest transmission timestamp in RACK.xmit_ts and its highest sequence number in RACK.end_seq. These two variables are used, in later steps, to estimate if some segments not yet delivered were likely lost. Given the information provided in an ACK, each segment cumulatively ACKed or SACKed is marked as delivered in the scoreboard. Since an ACK can also acknowledge retransmitted data segments, and retransmissions can be spurious, the sender needs to take care to avoid spurious inferences. For example, if the sender were to use timing information from a spurious retransmission, the RACK.rtt could be vastly underestimated.

To avoid spurious inferences, ignore a segment as invalid if any of its sequence range has been retransmitted before and either of two conditions is true:

1. The Timestamp Echo Reply field (TSecr) of the ACK's timestamp option [RFC7323], if available, indicates the ACK was not acknowledging the last retransmission of the segment.
2. The segment was last retransmitted less than RACK.min_rtt ago.

The second check is a heuristic when the TCP Timestamp option is not available, or when the round trip time is less than the TCP Timestamp clock granularity.

Among all the segments newly ACKed or SACKed by this ACK that pass the checks above, update the RACK.rtt to be the RTT sample calculated using this ACK. Furthermore, record the most recent Segment.xmit_ts in RACK.xmit_ts if it is ahead of RACK.xmit_ts. If Segment.xmit_ts equals RACK.xmit_ts (e.g. due to clock granularity limits) then compare Segment.end_seq and RACK.end_seq to break the tie when deciding whether to update the RACK.segment's associated state.

Step 2 may be summarized in pseudocode as:

```
RACK_sent_after(t1, seq1, t2, seq2):
    If t1 > t2:
        Return true
    Else if t1 == t2 AND seq1 > seq2:
        Return true
    Else:
        Return false

RACK_update():
    For each Segment newly acknowledged, cumulatively or selectively,
    in ascending order of Segment.xmit_ts:
        rtt = Now() - Segment.xmit_ts
        If Segment.retransmitted is TRUE:
            If ACK.ts_option.echo_reply < Segment.xmit_ts:
                Continue
            If rtt < RACK.min_rtt:
                Continue

        RACK.rtt = rtt
        If RACK_sent_after(Segment.xmit_ts, Segment.end_seq,
                           RACK.xmit_ts, RACK.end_seq):
            RACK.xmit_ts = Segment.xmit_ts
            RACK.end_seq = Segment.end_seq
```

Step 3: Detect data segment reordering

To detect reordering, the sender looks for original data segments being delivered out of order. To detect such cases, the sender tracks the highest sequence selectively or cumulatively acknowledged in the RACK.fack variable. The name "fack" stands for the most "Forward ACK" (this term is adopted from [FACK]). If a never-retransmitted segment that's below RACK.fack is (selectively or cumulatively) acknowledged, it has been delivered out of order. The sender sets RACK.reordering_seen to TRUE if such segment is identified.

```
RACK_detect_reordering():  
    For each Segment newly acknowledged, cumulatively or selectively,  
    in ascending order of Segment.end_seq:  
        If Segment.end_seq > RACK.fack:  
            RACK.fack = Segment.end_seq  
        Else if Segment.end_seq < RACK.fack AND  
            Segment.retransmitted is FALSE:  
            RACK.reordering_seen = TRUE
```

Step 4: Update RACK reordering window

The RACK reordering window, RACK.reo_wnd, serves as an adaptive allowance for settling time before marking a segment lost. This step documents a detailed algorithm that follows the principles outlined in the ``Reordering window adaptation'' section.

If no reordering has been observed, based on the previous step, then one way the sender can enter Fast Recovery is when the number of SACKed segments matches or exceeds DupThresh (similar to RFC6675). Furthermore, when no reordering has been observed the RACK.reo_wnd is set to 0 both upon entering and during Fast Recovery or RTO recovery.

Otherwise, if some reordering has been observed, then RACK does not trigger Fast Recovery based on DupThresh.

Whether or not reordering has been observed, RACK uses the reordering window to assess whether any segments can be marked lost. As a consequence, the sender also enters Fast Recovery when there are any number of SACKed segments as long as the reorder window has passed for some non-SACKed segments.

When the reordering window is not set to 0, it starts with a conservative RACK.reo_wnd of RACK.min_RTT/4. This value was chosen because Linux TCP used the same factor in its implementation to delay Early Retransmit [RFC5827] to reduce spurious loss detections in the presence of reordering, and experience showed this worked reasonably well [DMCG11].

However, the reordering detection in the previous step, Step 3, has a self-reinforcing drawback when the reordering window is too small to cope with the actual reordering. When that happens, RACK could spuriously mark reordered segments lost, causing them to be retransmitted. In turn, the retransmissions can prevent the necessary conditions for Step 3 to detect reordering, since this mechanism requires ACKs or SACKs for only segments that have never been retransmitted. In some cases such scenarios can persist, causing RACK to continue to spuriously mark segments lost without realizing the reordering window is too small.

To avoid the issue above, RACK dynamically adapts to higher degrees of reordering using DSACK options from the receiver. Receiving an ACK with a DSACK option indicates a possible spurious retransmission, suggesting that RACK.reo_wnd may be too small. The RACK.reo_wnd increases linearly for every round trip in which the sender receives some DSACK option, so that after N round trips in which a DSACK is received, the RACK.reo_wnd becomes $(N+1) * \text{min_RTT} / 4$, with an upper-bound of SRTT.

If the reordering is temporary then a large adapted reordering window would unnecessarily delay loss recovery later. Therefore, RACK persists using the inflated RACK.reo_wnd for up to 16 loss recoveries, after which it resets RACK.reo_wnd to its starting value, $\text{min_RTT} / 4$. The downside of resetting the reordering window is the risk of triggering spurious fast recovery episodes if the reordering remains high. The rationale for this approach is to bound such spurious recoveries to approximately once every 16 recoveries (less than 7%).

To track the linear scaling factor for the adaptive reordering window, RACK uses the variable RACK.reo_wnd_mult, which is initialized to 1 and adapts with observed reordering.

The following pseudocode implements the above algorithm for updating the RACK reordering window:

RACK_update_reo_wnd():

```

/* DSACK-based reordering window adaptation */
If RACK.dsack_round is not None AND
  SND.UNA >= RACK.dsack_round:
  RACK.dsack_round = None
/* Grow the reordering window per round that sees DSACK.
Reset the window after 16 DSACK-free recoveries */
If RACK.dsack_round is None AND
  any DSACK option is present on latest received ACK:
  RACK.dsack_round = SND.NXT
  RACK.reo_wnd_mult += 1
  RACK.reo_wnd_persist = 16
Else if exiting Fast or RTO recovery:
  RACK.reo_wnd_persist -= 1
  If RACK.reo_wnd_persist <= 0:
    RACK.reo_wnd_mult = 1

If RACK.reordering_seen is FALSE:
  If in Fast or RTO recovery:
    Return 0
  Else if RACK.segs_sacked >= DupThresh:
    Return 0
Return min(RACK.reo_wnd_mult * RACK.min_RTT / 4, SRTT)

```

Step 5: Detect losses.

For each segment that has not been SACKed, RACK considers that segment lost if another segment that was sent later has been delivered, and the reordering window has passed. RACK considers the reordering window to have passed if the RACK.segment was sent sufficiently after the segment in question, or a sufficient time has elapsed since the RACK.segment was S/ACKed, or some combination of the two. More precisely, RACK marks a segment lost if:

```

RACK.xmit_ts >= Segment.xmit_ts
AND
RACK.xmit_ts - Segment.xmit_ts + (now - RACK.ack_ts) >= RACK.reo_wnd

```

Solving this second condition for "now", the moment at which a segment is marked lost, yields:

```

now >= Segment.xmit_ts + RACK.reo_wnd + (RACK.ack_ts - RACK.xmit_ts)

```

Then (RACK.ack_ts - RACK.xmit_ts) is the round trip time of the most recently (re)transmitted segment that's been delivered. When segments are delivered in order, the most recently (re)transmitted segment that's been delivered is also the most recently delivered,

hence $\text{RACK.rtt} == \text{RACK.ack_ts} - \text{RACK.xmit_ts}$. But if segments were reordered, then the segment delivered most recently was sent before the most recently (re)transmitted segment. Hence $\text{RACK.rtt} > (\text{RACK.ack_ts} - \text{RACK.xmit_ts})$.

Since $\text{RACK.RTT} \geq (\text{RACK.ack_ts} - \text{RACK.xmit_ts})$, the previous equation reduces to saying that the sender can declare a segment lost when:

$$\text{now} \geq \text{Segment.xmit_ts} + \text{RACK.reo_wnd} + \text{RACK.rtt}$$

In turn, that is equivalent to stating that a RACK sender should declare a segment lost when:

$$\text{Segment.xmit_ts} + \text{RACK.rtt} + \text{RACK.reo_wnd} - \text{now} \leq 0$$

Note that if the value on the left hand side is positive, it represents the remaining wait time before the segment is deemed lost. But this risks a timeout (RTO) if no more ACKs come back (e.g., due to losses or application-limited transmissions) to trigger the marking. For timely loss detection, the sender is RECOMMENDED to install a reordering timer. This timer expires at the earliest moment when RACK would conclude that all the unacknowledged segments within the reordering window were lost.

The following pseudocode implements the algorithm above. When an ACK is received or the RACK reordering timer expires, call `RACK_detect_loss_and_arm_timer()`. The algorithm breaks timestamp ties by using the TCP sequence space, since high-speed networks often have multiple segments with identical timestamps.


```
RACK_detect_loss():
    timeout = 0
    RACK.reo_wnd = RACK_update_reo_wnd()
    For each segment, Segment, not acknowledged yet:
        If RACK_sent_after(RACK.xmit_ts, RACK.end_seq,
                           Segment.xmit_ts, Segment.end_seq):
            remaining = Segment.xmit_ts + RACK.rtt +
                        RACK.reo_wnd - Now()
            If remaining <= 0:
                Segment.lost = TRUE
                Segment.xmit_ts = INFINITE_TS
            Else:
                timeout = max(remaining, timeout)
    Return timeout

RACK_detect_loss_and_arm_timer():
    timeout = RACK_detect_loss()
    If timeout != 0
        Arm the RACK timer to call
        RACK_detect_loss_and_arm_timer() after timeout
```

As an optimization, an implementation can choose to check only segments that have been sent before `RACK.xmit_ts`. This can be more efficient than scanning the entire SACK scoreboard, especially when there are many segments in flight. The implementation can use a separate doubly-linked list ordered by `Segment.xmit_ts` and inserts a segment at the tail of the list when it is (re)transmitted, and removes a segment from the list when it is delivered or marked lost. In Linux TCP this optimization improved CPU usage by orders of magnitude during some fast recovery episodes on high-speed WAN networks.

6.3. Upon RTO expiration

Upon RTO timer expiration, RACK marks the first outstanding segment as lost (since it was sent an RTO ago); for all the other segments RACK only marks the segment lost if the time elapsed since the segment was transmitted is at least the sum of the recent RTT and the reordering window.

```
RACK_mark_losses_on_RTO():
    For each segment, Segment, not acknowledged yet:
        If SEG.SEQ == SND.UNA OR
           Segment.xmit_ts + RACK.rtt + RACK.reo_wnd - Now() <= 0:
            Segment.lost = TRUE
```

7. TLP Algorithm Details

7.1. Initializing state

Reset `TLP.is_retrans` and `TLP.end_seq` when initiating a connection, fast recovery, or RTO recovery.

```
TLP_init():  
    TLP.end_seq = None  
    TLP.is_retrans = false
```

7.2. Scheduling a loss probe

The sender schedules a loss probe timeout (PTO) to transmit a segment during the normal transmission process. The sender **SHOULD** start or restart a loss probe PTO timer after transmitting new data (that was not itself a loss probe) or upon receiving an ACK that cumulatively acknowledges new data, unless it is already in fast recovery, RTO recovery, or the sender has segments delivered out-of-order (i.e. `RACK.segs_sacked` is not zero). These conditions are excluded because they are addressed by similar mechanisms, like Limited Transmit [RFC3042], the RACK reordering timer, and F-RTO [RFC5682].

The sender calculates the PTO interval by taking into account a number of factors.

First, the default PTO interval is $2 \times \text{SRTT}$. By that time, it is prudent to declare that an ACK is overdue, since under normal circumstances, i.e. no losses, an ACK typically arrives in one SRTT. Choosing PTO to be exactly an SRTT would risk causing spurious probes, given that network and end-host delay variance can cause an ACK to be delayed beyond SRTT. Hence the PTO is conservatively chosen to be the next integral multiple of SRTT.

Second, when there is no SRTT estimate available, the PTO **SHOULD** be 1 second. This conservative value corresponds to the RTO value when no SRTT is available, per [RFC6298].

Third, when `FlightSize` is one segment, the sender **MAY** inflate PTO by `TLP.max_ack_delay` to accommodate a potential delayed acknowledgment and reduce the risk of spurious retransmissions. The actual value of `TLP.max_ack_delay` is implementation-specific.

Finally, if the time at which an RTO would fire (here denoted "`TCP_RTO_expiration()`") is sooner than the computed time for the PTO, then the sender schedules a TLP to be sent at that RTO time.

Summarizing these considerations in pseudocode form, a sender SHOULD use the following logic to select the duration of a PTO:

```
TLP_calc_PTO():
  If SRTT is available:
    PTO = 2 * SRTT
    If FlightSize is one segment:
      PTO += TLP.max_ack_delay
  Else:
    PTO = 1 sec

  If Now() + PTO > TCP_RTO_expiration():
    PTO = TCP_RTO_expiration() - Now()
```

7.3. Sending a loss probe upon PTO expiration

When the PTO timer expires, the sender MUST check whether both of the following conditions are met before sending a loss probe:

1. First, there is no other previous loss probe still in flight. This ensures that at any given time the sender has at most one additional packet in flight beyond the congestion window limit. This invariant is maintained using the state variable `TLP.end_seq`, which indicates the latest unacknowledged TLP loss probe's ending sequence. It is reset when the loss probe has been acknowledged or is deemed lost or irrelevant.
2. Second, the sender has obtained an RTT measurement since the last loss probe was transmitted, or, if the sender has not yet sent a loss probe on this connection, since the start of the connection. This condition ensures that loss probe retransmissions do not prevent taking the RTT samples necessary to adapt SRTT to an increase in path RTT.

If either one of these two conditions is not met, then the sender MUST skip sending a loss probe, and MUST proceed to re-arm the RTO timer, as specified at the end of this section.

If both conditions are met, then the sender SHOULD transmit a previously unsent data segment, if one exists and the receive window allows, and increment the FlightSize accordingly. Note that FlightSize could be one packet greater than the congestion window temporarily until the next ACK arrives.

If such an unsent segment is not available, then the sender SHOULD retransmit the highest-sequence segment sent so far and set `TLP.is_retrans` to true. This segment is chosen to deal with the retransmission ambiguity problem in TCP. Suppose a sender sends N

segments, and then retransmits the last segment (segment N) as a loss probe, and then the sender receives a SACK for segment N. As long as the sender waits for the RACK reordering window to expire, it doesn't matter if that SACK was for the original transmission of segment N or the TLP retransmission; in either case the arrival of the SACK for segment N provides evidence that the N-1 segments preceding segment N were likely lost.

In the case where there is only one original outstanding segment of data (N=1), the same logic (trivially) applies: an ACK for a single outstanding segment tells the sender the N-1=0 segments preceding that segment were lost. Furthermore, whether there are N>1 or N=1 outstanding segments, there is a question about whether the original last segment or its TLP retransmission were lost; the sender estimates whether there was such a loss using TLP recovery detection (see below).

The sender MUST follow the RACK transmission procedures in the "Upon Transmitting a Data Segment" section (see above) upon sending either a retransmission or new data loss probe. This is critical for detecting losses using the ACK for the loss probe.

After attempting to send a loss probe, regardless of whether a loss probe was sent, the sender MUST re-arm the RTO timer, not the PTO timer, if FlightSize is not zero. This ensures RTO recovery remains the last resort if TLP fails. The following pseudo code summarizes the operations.

TLP_send_probe() :

```
If TLP.end_seq is None and
  Sender has taken a new RTT sample since last probe or
  the start of connection:
  TLP.is_retrans = false
  Segment = send buffer segment starting at SND.NXT
  If Segment exists and fits the peer receive window limit:
    /* Transmit the lowest-sequence unsent Segment */
    Transmit Segment
    RACK_transmit_data(Segment)
    TLP.end_seq = SND.NXT
    Increase FlightSize by Segment length
  Else:
    /* Retransmit the highest-sequence Segment sent */
    Segment = send buffer segment ending at SND.NXT
    Transmit Segment
    RACK_retransmit_data(Segment)
    TLP.end_seq = SND.NXT
```

7.4. Detecting losses using the ACK of the loss probe

When there is packet loss in a flight ending with a loss probe, the feedback solicited by a loss probe will reveal one of two scenarios, depending on the pattern of losses.

7.4.1. General case: detecting packet losses using RACK

If the loss probe and the ACK that acknowledges the probe are delivered successfully, RACK-TLP uses this ACK -- just as it would with any other ACK -- to detect if any segments sent prior to the probe were dropped. RACK would typically infer that any unacknowledged data segments sent before the loss probe were lost, since they were sent sufficiently far in the past (at least one PTO has elapsed, plus one round-trip for the loss probe to be ACKed). More specifically, `RACK_detect_loss()` (step 5) would mark those earlier segments as lost. Then the sender would trigger a fast recovery to recover those losses.

7.4.2. Special case: detecting a single loss repaired by the loss probe

If the TLP retransmission repairs all the lost in-flight sequence ranges (i.e. only the last segment in the flight was lost), the ACK for the loss probe appears to be a regular cumulative ACK, which would not normally trigger the congestion control response to this packet loss event. The following TLP recovery detection mechanism examines ACKs to detect this special case to make congestion control respond properly [RFC5681].

After a TLP retransmission, the sender checks for this special case of a single loss that is recovered by the loss probe itself. To accomplish this, the sender checks for a duplicate ACK or DSACK indicating that both the original segment and TLP retransmission arrived at the receiver, meaning there was no loss. If the TLP sender does not receive such an indication, then it MUST assume that either the original data segment, the TLP retransmission, or a corresponding ACK were lost, for congestion control purposes.

If the TLP retransmission is spurious, a receiver that uses DSACK would return an ACK that covers `TLP.end_seq` with a DSACK option (Case 1). If the receiver does not support DSACK, it would return a DUPACK without any SACK option (Case 2). If the sender receives an ACK matching either case, then the sender estimates that the receiver received both the original data segment and the TLP probe retransmission, and so the sender considers the TLP episode to be done, and records that fact by setting `TLP.end_seq` to None.

Upon receiving an ACK that covers some sequence number after TLP.end_seq, the sender should have received any ACKs for the original segment and TLP probe retransmission segment. At that time, if the TLP.end_seq is still set, and thus indicates that the TLP probe retransmission remains unacknowledged, then the sender should presume that at least one of its data segments was lost. The sender then SHOULD invoke a congestion control response equivalent to a fast recovery.

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

TLP_process_ack(ACK):

```
If TLP.end_seq is not None AND ACK's ack. number >= TLP.end_seq:
    If not TLP.is_retrans:
        TLP.end_seq = None      /* TLP of new data delivered */
    Else if ACK has a DSACK option matching TLP.end_seq:
        TLP.end_seq = None      /* Case 1, above */
    Else If ACK's ack. number > TLP.end_seq:
        TLP.end_seq = None      /* Repaired the single loss */
        (Invoke congestion control to react to
         the loss event the probe has repaired)
    Else If ACK is a DUPACK without any SACK option:
        TLP.end_seq = None      /* Case 2, above */
```

8. Managing RACK-TLP timers

The RACK reordering timer, the TLP PTO timer, the RTO, and Zero Window Probe (ZWP) timer [RFC793] are mutually exclusive and used in different scenarios. When arming a RACK reordering timer or TLP PTO timer, the sender SHOULD cancel any other pending timer(s). An implementation is expected to have one timer with an additional state variable indicating the type of the timer.

9. Discussion

9.1. Advantages and disadvantages

The biggest advantage of RACK-TLP is that every data segment, whether it is an original data transmission or a retransmission, can be used to detect losses of the segments sent chronologically prior to it. This enables RACK-TLP to use fast recovery in cases with application-limited flights of data, lost retransmissions, or data segment reordering events. Consider the following examples:

1. Packet drops at the end of an application data flight: Consider a sender that transmits an application-limited flight of three data segments (P1, P2, P3), and P1 and P3 are lost. Suppose the transmission of each segment is at least RACK.reo_wnd after the

transmission of the previous segment. RACK will mark P1 as lost when the SACK of P2 is received, and this will trigger the retransmission of P1 as R1. When R1 is cumulatively acknowledged, RACK will mark P3 as lost and the sender will retransmit P3 as R3. This example illustrates how RACK is able to repair certain drops at the tail of a transaction without an RTO recovery. Notice that neither the conventional duplicate ACK threshold [RFC5681], nor [RFC6675], nor the Forward Acknowledgment [FACK] algorithm can detect such losses, because of the required segment or sequence count.

2. Lost retransmission: Consider a flight of three data segments (P1, P2, P3) that are sent; P1 and P2 are dropped. Suppose the transmission of each segment is at least RACK.reo_wnd after the transmission of the previous segment. When P3 is SACKed, RACK will mark P1 and P2 lost and they will be retransmitted as R1 and R2. Suppose R1 is lost again but R2 is SACKed; RACK will mark R1 lost and trigger retransmission again. Again, neither the conventional three duplicate ACK threshold approach, nor [RFC6675], nor the Forward Acknowledgment [FACK] algorithm can detect such losses. And such a lost retransmission can happen when TCP is being rate-limited, particularly by token bucket policers with large bucket depth and low rate limit; in such cases retransmissions are often lost repeatedly because standard congestion control requires multiple round trips to reduce the rate below the policed rate.
3. Packet reordering: Consider a simple reordering event where a flight of segments are sent as (P1, P2, P3). P1 and P2 carry a full payload of MSS octets, but P3 has only a 1-octet payload. Suppose the sender has detected reordering previously and thus RACK.reo_wnd is min_RTT/4. Now P3 is reordered and delivered first, before P1 and P2. As long as P1 and P2 are delivered within min_RTT/4, RACK will not consider P1 and P2 lost. But if P1 and P2 are delivered outside the reordering window, then RACK will still spuriously mark P1 and P2 lost.

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

RACK-TLP works more efficiently with TCP Segmentation Offload (TSO) compared to DUPACK-counting. RACK always marks the entire TSO aggregate lost because the segments in the same TSO aggregate have the same transmission timestamp. By contrast, the algorithms based on sequence counting (e.g., [RFC6675] [RFC5681]) may mark only a

subset of segments in the TSO aggregate lost, forcing the stack to perform expensive fragmentation of the TSO aggregate, or to selectively tag individual segments lost in the scoreboard.

The main drawback of RACK-TLP is the additional states required compared to DUPACK-counting. RACK requires the sender to record the transmission time of each segment sent at a clock granularity that is finer than 1/4 of the minimum RTT of the connection. TCP implementations that record this already for RTT estimation do not require any new per-packet state. But implementations that are not yet recording segment transmission times will need to add per-packet internal state (expected to be either 4 or 8 octets per segment or TSO aggregate) to track transmission times. In contrast, [RFC6675] loss detection approach does not require any per-packet state beyond the SACK scoreboard; this is particularly useful on ultra-low RTT networks where the RTT may be less than the sender TCP clock granularity (e.g. inside data-centers). Another disadvantage is the reordering timer may expire prematurely (like any other retransmission timer) to cause higher spurious retransmission especially if DSACK is not supported.

9.2. Relationships with other loss recovery algorithms

The primary motivation of RACK-TLP is to provide a general alternative to some of the standard loss recovery algorithms [RFC5681] [RFC6675] [RFC5827] [RFC4653]. In particular, [RFC6675] is not designed to handle lost retransmissions, so its NextSeg() does not work for lost retransmissions and it does not specify the corresponding required additional congestion response. Therefore, [RFC6675] MUST NOT be used with RACK-TLP; instead, a modified recovery algorithm that carefully addresses such a case is needed.

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

RACK-TLP is compatible with the standard RTO [RFC6298], RTO-restart [RFC7765], F-RTO [RFC5682] and Eifel algorithms [RFC3522]. This is because RACK-TLP only detects loss by using ACK events. It neither changes the RTO timer calculation nor detects spurious RTO. RACK-TLP

slightly changes the behavior of [RFC6298] by preceding the RTO with a TLP and reducing potential spurious retransmissions after RTO.

9.3. Interaction with congestion control

RACK-TLP intentionally decouples loss detection from congestion control. RACK-TLP only detects losses; it does not modify the congestion control algorithm [RFC5681] [RFC6937]. A segment marked lost by RACK-TLP MUST NOT be retransmitted until congestion control deems this appropriate. As mentioned in the caption for Figure 1, [RFC5681] mandates a principle that loss in two successive windows of data, or the loss of a retransmission, must be taken as two indications of congestion and therefore trigger two separate reactions. [RFC6937] is RECOMMENDED for the specific congestion control actions taken upon the losses detected by RACK-TLP. In the absence of PRR, when RACK-TLP detects a lost retransmission the congestion control MUST trigger an additional congestion response per the aforementioned principle in [RFC5681]. If multiple original transmissions or retransmissions were lost in a window, the congestion control specified in [RFC5681] only reacts once per window. The congestion control implementer is advised to carefully consider this subtle situation introduced by RACK-TLP.

The only exception -- the only way in which RACK-TLP modulates the congestion control algorithm -- is that one outstanding loss probe can be sent even if the congestion window is fully used. However, this temporary over-commit is accounted for and credited in the in-flight data tracked for congestion control, so that congestion control will erase the over-commit upon the next ACK.

If packet losses happen after reordering has been observed, RACK-TLP may take longer to detect losses than the pure DUPACK-counting approach. In this case TCP may continue to increase the congestion window upon receiving ACKs during this time, making the sender more aggressive.

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

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

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

The difference in recovery latency ($\text{RTO} + 4 \cdot \text{RTT}$ vs $6 \cdot \text{RTT}$) can be significant if the RTT is much smaller than the minimum RTO (1 second in [RFC6298]) or if the RTT is large. The former case can happen in local area networks, data-center networks, or content distribution networks with deep deployments. The latter case can happen in developing regions with highly congested and/or high-latency networks.

9.4. TLP recovery detection with delayed ACKs

Delayed or stretched ACKs complicate the detection of repairs done by TLP, since with such ACKs the sender takes a longer time to receive fewer ACKs than would normally be expected. To mitigate this complication, before sending a TLP loss probe retransmission, the sender should attempt to wait long enough that the receiver has sent any delayed ACKs that it is withholding. The sender algorithm described above features such a delay, in the form of `TLP.max_ack_delay`. Furthermore, if the receiver supports DSACK then in the case of a delayed ACK the sender's TLP recovery detection mechanism (see above) can use the DSACK information to infer that the original and TLP retransmission both arrived at the receiver.

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

9.5. RACK-TLP for other transport protocols

RACK-TLP can be implemented in other transport protocols (e.g., [QUIC-LR]). The [Sprout] loss detection algorithm was also independently designed to use a 10ms reordering window to improve its loss detection similar to RACK.

10. Security Considerations

RACK-TLP algorithm behavior is based on information conveyed in SACK options, so it has security considerations similar to those described in the Security Considerations section of [RFC6675].

Additionally, RACK-TLP has a lower risk profile than [RFC6675] because it is not vulnerable to ACK-splitting attacks [SCWA99]: for an MSS-size segment sent, the receiver or the attacker might send MSS ACKs that SACK or acknowledge one additional byte per ACK. This would not fool RACK. In such a scenario, RACK.xmit_ts would not advance, because all the sequence ranges within the segment were transmitted at the same time, and thus carry the same transmission timestamp. In other words, SACKing only one byte of a segment or SACKing the segment in entirety have the same effect with RACK.

11. IANA Considerations

This document makes no request of IANA.

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

12. Acknowledgments

The authors thank Matt Mathis for his insights in FACK and Michael Welzl for his per-packet timer idea that inspired this work. Eric Dumazet, Randy Stewart, Van Jacobson, Ian Swett, Rick Jones, Jana Iyengar, Hiren Panchasara, Praveen Balasubramanian, Yoshifumi Nishida, Bob Briscoe, Felix Weinrank, Michael Tuexen, Martin Duke, Ilpo Jarvinen, Theresa Enghardt, Mirja Kuehlewind, Gorrry Fairhurst, Markku Kojo, and Yi Huang contributed to this document or the implementations in Linux, FreeBSD, Windows, and QUIC.

13. References

13.1. Normative References

- [RFC2018] Mathis, M. and J. Mahdavi, "TCP Selective Acknowledgment Options", RFC 2018, October 1996.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, March 1997.
- [RFC2883] Floyd, S., Mahdavi, J., Mathis, M., and M. Podolsky, "An Extension to the Selective Acknowledgement (SACK) Option for TCP", RFC 2883, July 2000.

- [RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, September 2009.
- [RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, June 2011.
- [RFC6675] Blanton, E., Allman, M., Wang, L., Jarvinen, I., Kojo, M., and Y. Nishida, "A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP", RFC 6675, August 2012.
- [RFC7323] Borman, D., Braden, B., Jacobson, V., and R. Scheffenegger, "TCP Extensions for High Performance", September 2014.
- [RFC793] Postel, J., "Transmission Control Protocol", September 1981.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", May 2017.

13.2. Informative References

- [DMCG11] Dukkkipati, N., Matthis, M., Cheng, Y., and M. Ghobadi, "Proportional Rate Reduction for TCP", ACM SIGCOMM Conference on Internet Measurement , 2011.
- [FACK] Mathis, M. and M. Jamshid, "Forward acknowledgement: refining TCP congestion control", ACM SIGCOMM Computer Communication Review, Volume 26, Issue 4, Oct. 1996. , 1996.
- [POLICER16] Flach, T., Papageorge, P., Terzis, A., Pedrosa, L., Cheng, Y., Karim, T., Katz-Bassett, E., and R. Govindan, "An Analysis of Traffic Policing in the Web", ACM SIGCOMM , 2016.
- [QUIC-LR] Iyengar, J. and I. Swett, "QUIC Loss Detection and Congestion Control", draft-ietf-quic-recovery (work in progress), October 2020.
- [RFC3042] Allman, M., Balakrishnan, H., and S. Floyd, "Enhancing TCP's Loss Recovery Using Limited Transmit", January 2001.
- [RFC3522] Ludwig, R. and M. Meyer, "The Eifel Detection Algorithm for TCP", April 2003.

- [RFC4653] Bhandarkar, S., Reddy, A., Allman, M., and E. Blanton, "Improving the Robustness of TCP to Non-Congestion Events", August 2006.
- [RFC5682] Sarolahti, P., Kojo, M., Yamamoto, K., and M. Hata, "Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP", RFC 5682, September 2009.
- [RFC5827] Allman, M., Ayesta, U., Wang, L., Blanton, J., and P. Hurtig, "Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)", RFC 5827, April 2010.
- [RFC6937] Mathis, M., Dukkkipati, N., and Y. Cheng, "Proportional Rate Reduction for TCP", May 2013.
- [RFC7765] Hurtig, P., Brunstrom, A., Petlund, A., and M. Welzl, "TCP and SCTP RTO Restart", February 2016.
- [SCWA99] Savage, S., Cardwell, N., Wetherall, D., and T. Anderson, "TCP Congestion Control With a Misbehaving Receiver", ACM Computer Communication Review, 29(5) , 1999.
- [Sprout] Winstein, K., Sivaraman, A., and H. Balakrishnan, "Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks", USENIX Symposium on Networked Systems Design and Implementation (NSDI) , 2013.

Authors' Addresses

Yuchung Cheng
Google, Inc

Email: ycheng@google.com

Neal Cardwell
Google, Inc

Email: ncardwell@google.com

Nandita Dukkkipati
Google, Inc

Email: nanditad@google.com

Priyaranjan Jha
Google, Inc

Email: priyarjha@google.com

Internet Engineering Task Force
Internet-Draft
Obsoletes: 793, 879, 2873, 6093, 6429, 6528,
6691 (if approved)
Updates: 5961, 1011, 1122 (if approved)
Intended status: Standards Track
Expires: 8 September 2022

W. Eddy, Ed.
MTI Systems
7 March 2022

Transmission Control Protocol (TCP) Specification
draft-ietf-tcpm-rfc793bis-28

Abstract

This document specifies the Transmission Control Protocol (TCP). TCP is an important transport layer protocol in the Internet protocol 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 RFCs 879, 2873, 6093, 6429, 6528, and 6691 that updated parts of RFC 793. It updates RFCs 1011 and 1122, and should be considered as a replacement for the portions of those document dealing with TCP requirements. It also updates RFC 5961 by adding a small clarification in reset handling while in the SYN-RECEIVED state. The TCP header control bits from RFC 793 have also been updated based on RFC 3168.

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.

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 8 September 2022.

Copyright Notice

Copyright (c) 2022 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 Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised 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	4
2. Introduction	5
2.1. Requirements Language	5
2.2. Key TCP Concepts	6
3. Functional Specification	6
3.1. Header Format	6
3.2. Specific Option Definitions	12
3.2.1. Other Common Options	13
3.2.2. Experimental TCP Options	13
3.3. TCP Terminology Overview	13
3.3.1. Key Connection State Variables	13
3.3.2. State Machine Overview	15
3.4. Sequence Numbers	18
3.4.1. Initial Sequence Number Selection	21
3.4.2. Knowing When to Keep Quiet	23
3.4.3. The TCP Quiet Time Concept	23
3.5. Establishing a connection	25
3.5.1. Half-Open Connections and Other Anomalies	28
3.5.2. Reset Generation	31
3.5.3. Reset Processing	32

3.6.	Closing a Connection	32
3.6.1.	Half-Closed Connections	35
3.7.	Segmentation	35
3.7.1.	Maximum Segment Size Option	37
3.7.2.	Path MTU Discovery	38
3.7.3.	Interfaces with Variable MTU Values	39
3.7.4.	Nagle Algorithm	39
3.7.5.	IPv6 Jumbograms	40
3.8.	Data Communication	40
3.8.1.	Retransmission Timeout	41
3.8.2.	TCP Congestion Control	41
3.8.3.	TCP Connection Failures	42
3.8.4.	TCP Keep-Alives	43
3.8.5.	The Communication of Urgent Information	44
3.8.6.	Managing the Window	45
3.9.	Interfaces	50
3.9.1.	User/TCP Interface	50
3.9.2.	TCP/Lower-Level Interface	59
3.10.	Event Processing	61
3.10.1.	OPEN Call	63
3.10.2.	SEND Call	64
3.10.3.	RECEIVE Call	65
3.10.4.	CLOSE Call	67
3.10.5.	ABORT Call	68
3.10.6.	STATUS Call	69
3.10.7.	SEGMENT ARRIVES	70
3.10.8.	Timeouts	84
4.	Glossary	84
5.	Changes from RFC 793	89
6.	IANA Considerations	96
7.	Security and Privacy Considerations	97
8.	Acknowledgements	99
9.	References	100
9.1.	Normative References	100
9.2.	Informative References	102
Appendix A.	Other Implementation Notes	107
A.1.	IP Security Compartment and Precedence	108
A.1.1.	Precedence	108
A.1.2.	MLS Systems	109
A.2.	Sequence Number Validation	109
A.3.	Nagle Modification	109
A.4.	Low Watermark Settings	110
Appendix B.	TCP Requirement Summary	110
Author's Address	114

1. Purpose and Scope

In 1981, RFC 793 [16] 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 widely implemented, 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 core specification for TCP [50]. Over time, a number of errata have been filed against RFC 793. There have also been deficiencies found and resolved in security, performance, and many other aspects. The number of enhancements has grown over time across many separate documents. These were never accumulated together into a comprehensive update to the base specification.

The purpose of this document is to bring together all of the IETF Standards Track changes and other clarifications that have been made to the base TCP functional specification and unify them into an updated version of RFC 793.

Some companion documents are referenced for important algorithms that are used by TCP (e.g. for congestion control), but have not been completely included 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. This document focuses on the common basis all TCP implementations must support in order to interoperate. Since 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 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. This document preserves 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 purposes, 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, connection termination, and packet retransmission to repair losses.

This document describes the basic functionality expected in modern TCP implementations, and replaces the protocol specification in RFC 793. It does not replicate or attempt to update the introduction and philosophy content in Sections 1 and 2 of 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" [50] 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. [8]) 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 TCP implementations today include the high-performance extensions in [48], but these are not strictly required or discussed in this document. Multipath considerations for TCP are also specified separately in [59].

A list of changes from RFC 793 is contained in Section 5.

2.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [3][12] when, and only when, they appear in all capitals, as shown here.

Each use of RFC 2119 keywords in the document is individually labeled and referenced in Appendix B that summarizes implementation requirements.

Sentences using "MUST" are labeled as "MUST-X" with X being a numeric identifier enabling the requirement to be located easily when referenced from Appendix B.

Similarly, sentences using "SHOULD" are labeled with "SHLD-X", "MAY" with "MAY-X", and "RECOMMENDED" with "REC-X".

For the purposes of this labeling, "SHOULD NOT" and "MUST NOT" are labeled the same as "SHOULD" and "MUST" instances.

2.2. Key TCP Concepts

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

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

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

TCP supports unicast delivery of data. Anycast applications exist that successfully use TCP without modifications, though there is some risk of instability due to changes of lower-layer forwarding behavior [47].

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

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

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

A more detailed description of TCP features compared to other transport protocols can be found in Section 3.1 of [53]. Further description of the motivations for developing TCP and its role in the Internet protocol stack can be found in Section 2 of [16] and earlier versions of the TCP specification.

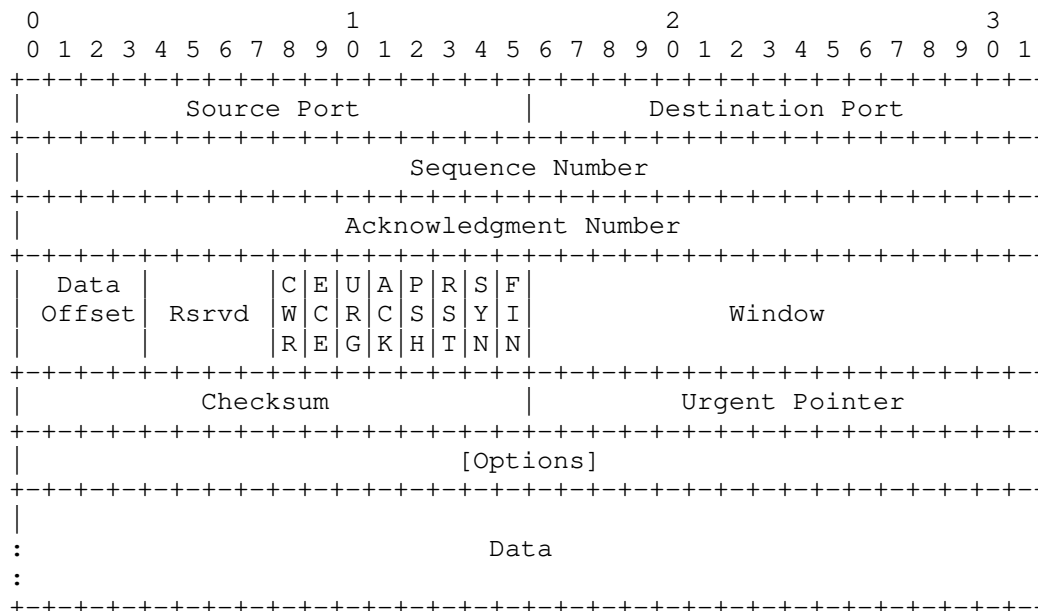
3. Functional Specification

3.1. Header Format

TCP segments are sent as internet datagrams. The Internet Protocol (IP) header carries several information fields, including the source and destination host addresses [1] [13]. A TCP header follows the IP headers, supplying information specific to the TCP protocol. This division allows for the existence of host level protocols other than TCP. In early development of the Internet suite of protocols, the IP header fields had been a part of TCP.

This document describes the TCP protocol. The TCP protocol uses TCP Headers.

A TCP Header, followed by any user data in the segment, is formatted as follows, using the style from [67]:



Note that one tick mark represents one bit position.

Figure 1: TCP Header Format

where:

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 the SYN flag is set). If SYN is set 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 (DOffset): 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 integer multiple of 32 bits long.

Reserved (Rsrvd): 4 bits.

A set of control bits reserved for future use. Must be zero in generated segments and must be ignored in received segments, if corresponding future features are unimplemented by the sending or receiving host.

The control bits are also known as "flags". Assignment is managed by IANA from the "TCP Header Flags" registry [63]. The currently assigned control bits are CWR, ECE, URG, ACK, PSH, RST, SYN, and FIN.

CWR: 1 bit.

Congestion Window Reduced (see [6]).

ECE: 1 bit.

ECN-Echo (see [6]).

URG: 1 bit.

Urgent Pointer field is significant.

ACK: 1 bit.

Acknowledgment field is significant.

PSH: 1 bit.

Push Function (see the Send Call description in Section 3.9.1).

RST: 1 bit.

Reset the connection.

SYN: 1 bit.

Synchronize sequence numbers.

FIN: 1 bit.

No more data from sender.

Window: 16 bits.

The number of data octets beginning with the one indicated in the acknowledgment field that the sender of this segment is willing to accept. The value is shifted when the Window Scaling extension is used [48].

The window size MUST be treated as an unsigned number, or else large window sizes will appear like negative windows and TCP will not work (MUST-1). It is RECOMMENDED that implementations will reserve 32-bit fields for the send and receive window sizes in the connection record and do all window computations with 32 bits (RECOMMENDED-1).

Checksum: 16 bits.

The checksum field is the 16 bit ones' complement of the ones' complement sum of all 16 bit words in the header and text. The checksum computation needs to ensure the 16-bit alignment of the data being summed. If a segment contains an odd number of header and text octets, alignment can be achieved by padding the last octet with zeros on its right 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 (Figure 2) conceptually prefixed to the TCP header. The pseudo header is 96 bits for IPv4 and 320 bits for IPv6. Including the pseudo header in the checksum gives the TCP connection protection against misrouted segments. This information is carried in IP headers and is transferred across the TCP/Network interface in the arguments or results of calls by the TCP implementation on the IP layer.

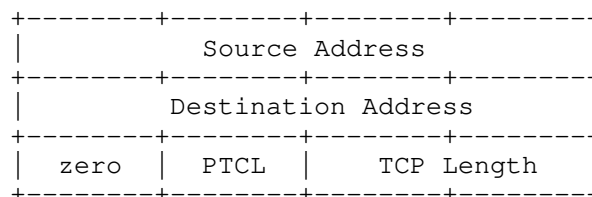


Figure 2: IPv4 Pseudo Header

Pseudo header components for IPv4:

Source Address: the IPv4 source address in network byte order

Destination Address: the IPv4 destination address in network byte order

zero: bits set to zero

PTCL: the protocol number from the IP header

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

For IPv6, the pseudo header is defined in Section 8.1 of RFC 8200 [13], and contains the IPv6 Source Address and Destination Address, an Upper Layer Packet Length (a 32-bit value otherwise equivalent to TCP Length in the IPv4 pseudo header), three bytes of zero-padding, and a Next Header value (differing from the IPv6 header value in the case of extension headers present in between IPv6 and TCP).

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

Urgent Pointer: 16 bits.

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

Options: [TCP Option]; size(Options) == (DOffset-5)*32; present only when DOffset > 5. Note that this size expression also includes any padding trailing the actual options present.

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 (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 of zeros (MUST-69).

The list of all currently defined options is managed by IANA [62], and each option is defined in other RFCs, as indicated there. That set includes experimental options that can be extended to support multiple concurrent usages [46].

A given TCP implementation can support any currently defined options, but the following options MUST be supported (MUST-4 - note Maximum Segment Size option support is also part of MUST-19 in Section 3.7.2):

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

These options are specified in detail in Section 3.2.

A TCP implementation MUST be able to receive a TCP option in any segment (MUST-5).

A TCP implementation MUST (MUST-6) ignore without error any TCP option it does not implement, assuming that the option has a length field. All TCP options except End of option list and No-Operation MUST have length fields, including all future options (MUST-68). TCP implementations MUST be prepared to handle an illegal option length (e.g., zero); a suggested procedure is to reset the connection and log the error cause (MUST-7).

Note: There is ongoing work to extend the space available for TCP options, such as [66].

Data: variable length.

User data carried by the TCP segment.

3.2. Specific Option Definitions

A TCP Option, in the mandatory option set, is one of: an End of Option List Option, a No-Operation Option, or a Maximum Segment Size Option.

An End of Option List Option is formatted as follows:

```

0
0 1 2 3 4 5 6 7
+---+---+---+---+---+---+
|           0           |
+---+---+---+---+---+---+

```

where:

Kind: 1 byte; 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.

A No-Operation Option is formatted as follows:

```

0
0 1 2 3 4 5 6 7
+---+---+---+---+---+---+
|           1           |
+---+---+---+---+---+---+

```

where:

Kind: 1 byte; Kind == 1.

This option code can 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 (MUST-64).

A Maximum Segment Size Option is formatted as follows:

```

0                               1                               2                               3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|           2           | Length | Maximum Segment Size (MSS) |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

where:

Kind: 1 byte; Kind == 2.

If this option is present, then it communicates the maximum receive segment size at the TCP endpoint that 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 (MUST-65). If this option is not used, any segment size is allowed. A more complete description of this option is provided in Section 3.7.1.

Length: 1 byte; Length == 4.

Length of the option in bytes.

Maximum Segment Size (MSS): 2 bytes.

The maximum receive segment size at the TCP endpoint that sends this segment.

3.2.1. Other Common Options

Additional RFCs define some other commonly used options that are recommended to implement for high performance, but not necessary for basic TCP interoperability. These are the TCP Selective Acknowledgement (SACK) option [23][27], TCP Timestamp (TS) option [48], and TCP Window Scaling (WS) option [48].

3.2.2. Experimental TCP Options

Experimental TCP option values are defined in [31], and [46] describes the current recommended usage for these experimental values.

3.3. TCP Terminology Overview

This section includes an overview of key terms needed to understand the detailed protocol operation in the rest of the document. There is a glossary of terms in Section 4.

3.3.1. Key Connection State Variables

Before we can discuss very much about the operation of the TCP implementation we need to introduce some detailed terminology. The maintenance of a TCP connection requires maintaining state for 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 IP addresses and port numbers, the IP security level and compartment of the

connection (see Appendix A.1), pointers to the user's send and receive buffers, pointers to the retransmit queue and to the current segment. In addition, several variables relating to the send and receive sequence numbers are stored in the TCB.

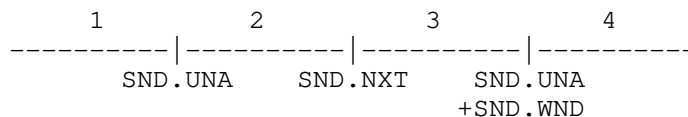
Send Sequence Variables:

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

Receive Sequence Variables:

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

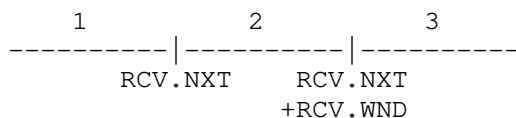
The following diagrams may help to relate some of these variables to the sequence space.



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

Figure 3: Send Sequence Space

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



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

Figure 4: Receive Sequence Space

The receive window is the portion of the sequence space labeled 2 in Figure 4.

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

Current Segment Variables:

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

3.3.2. State Machine Overview

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

LISTEN - represents waiting for a connection request from any remote TCP peer 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 peer, or an acknowledgment of the connection termination request previously sent.

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

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 peer.

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

TIME-WAIT - represents waiting for enough time to pass to be sure the remote TCP peer received the acknowledgment of its connection termination request, and to avoid new connections being impacted by delayed segments from previous connections.

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 OPEN call specifies whether connection establishment is to be actively pursued, or to be passively waited for.

A passive OPEN request means that the process wants to accept incoming connection requests, in contrast to an active OPEN attempting to initiate a connection.

The state diagram in Figure 5 illustrates only state changes, together with the causing events and resulting actions, but addresses neither error conditions nor actions that are not connected with state changes. In a later section, more detail is offered with respect to the reaction of the TCP implementation 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.

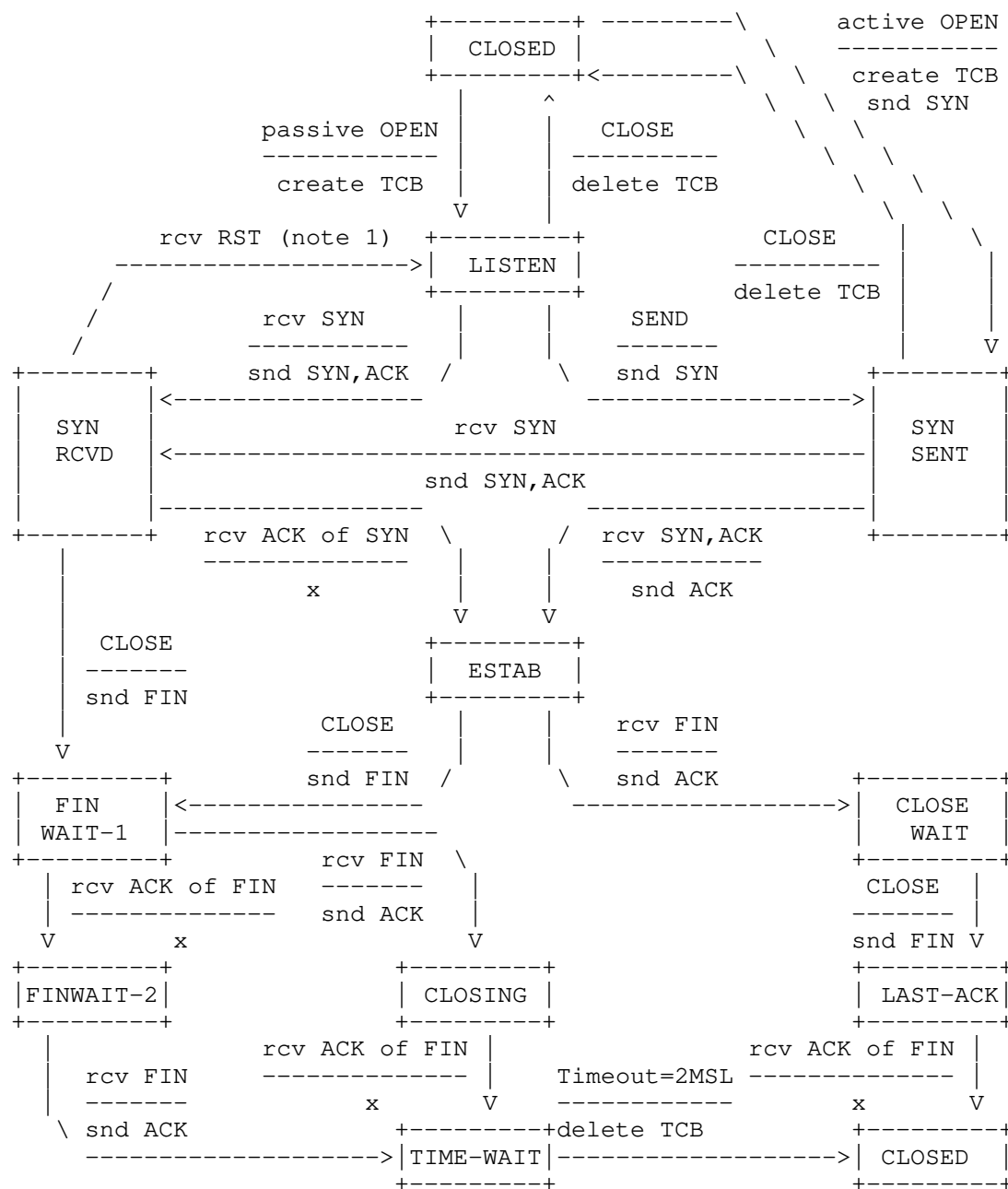


Figure 5: TCP Connection State Diagram

The following notes apply to Figure 5:

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: The figure omits a transition from FIN-WAIT-1 to TIME-WAIT if a FIN is received and the local FIN is also acknowledged.

Note 3: A RST can be sent from any state with a corresponding transition to TIME-WAIT (see [71] for rationale). These transitions are not explicitly shown, otherwise the diagram would become very difficult to read. Similarly, receipt of a RST from any state results in a transition to LISTEN or CLOSED, though this is also omitted from the diagram for legibility.

3.4. 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 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 that the TCP implementation 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 that are expected (i.e., that the segment "overlaps" the receive window).

In response to sending data the TCP endpoint 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 peer (next sequence number expected by the receiving TCP peer)

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 segment, 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	$\text{SEG.SEQ} = \text{RCV.NXT}$
0	>0	$\text{RCV.NXT} \leq \text{SEG.SEQ} < \text{RCV.NXT} + \text{RCV.WND}$
>0	0	not acceptable
>0	>0	$\text{RCV.NXT} \leq \text{SEG.SEQ} < \text{RCV.NXT} + \text{RCV.WND}$ or $\text{RCV.NXT} \leq \text{SEG.SEQ} + \text{SEG.LEN} - 1 < \text{RCV.NXT} + \text{RCV.WND}$

Note that when the receive window is zero no segments should be acceptable except ACK segments. Thus, it is possible for a TCP implementation to maintain a zero receive window while transmitting data and receiving ACKs. A TCP receiver MUST process the RST and URG fields of all incoming segments, even when the receive window is zero (MUST-66).

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.

3.4.1. Initial Sequence Number Selection

A connection is defined by a pair of sockets. Connections can be reused. 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 implementation 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 support this, the TIME-WAIT state limits the rate of connection reuse, while the initial sequence number selection described below further protects against ambiguity about what incarnation of a connection an incoming packet corresponds to.

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 endpoint loses all knowledge of the sequence numbers it has been using. When new connections are created, an initial sequence number (ISN) generator is employed that 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 [43].

TCP Initial Sequence Numbers are generated from a number sequence that monotonically increases until it wraps, known loosely as a "clock". This clock is a 32-bit counter that typically increments at least once every roughly 4 microseconds, although it is neither assumed to be realtime nor precise, and need not persist across reboots. The clock component is intended to ensure 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.

A TCP implementation **MUST** use the above type of "clock" for clock-driven selection of initial sequence numbers (**MUST-8**), 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") (SHLD-1). F() MUST NOT be computable from the outside (MUST-9), or an attacker could still guess at sequence numbers from the ISN used for some other connection. The PRF could be implemented as a cryptographic 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 [43].

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 peer, and the initial receive sequence number (IRS) is learned during the connection establishing procedure.

For a connection to be established or initialized, the two TCP peers 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 ISNs.

The synchronization requires each side to send its own initial sequence number and to receive a confirmation of it in acknowledgment from the remote TCP peer. Each side must also receive the remote peer'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 (3WHS).

A 3WHS is necessary because sequence numbers are not tied to a global clock in the network, and TCP implementations may have different mechanisms for picking the ISNs. The receiver of the first SYN has no way of knowing whether the segment was an old 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 for ISN selection are discussed in [70].

3.4.2. Knowing When to Keep Quiet

A theoretical problem exists where data could be corrupted due to confusion between old segments in the network and new ones after a host reboots, if the same port numbers and sequence space are reused. The "Quiet Time" concept discussed below addresses this and the discussion of it is included for situations where it might be relevant, although it is not felt to be necessary in most current implementations. The problem was more relevant earlier in the history of TCP. In practical use on the Internet today, the error-prone conditions are sufficiently unlikely that it is felt safe to ignore. Reasons why it is now negligible include: (a) ISS and ephemeral port randomization have reduced likelihood of reuse of port numbers and sequence numbers after reboots, (b) the effective MSL of the Internet has declined as links have become faster, and (c) reboots often taking longer than an MSL anyways.

To be sure that a TCP implementation does not create a segment carrying a sequence number that may be duplicated by an old segment remaining in the network, the TCP endpoint must keep quiet for an MSL before assigning any sequence numbers upon starting up or recovering from a situation where 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 endpoint 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.

3.4.3. The TCP Quiet Time Concept

Hosts that for any reason lose 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 that the host is a part of. 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 data by some receivers in the internet system.

TCP endpoints 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, TCP implementations 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 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 10s of megabits/sec. At 100 megabits/sec, the cycle time is 5.4 minutes, which may be a little short, but still within reason. Much higher data rates are possible today, with implications described in the final paragraph of this subsection.

The basic duplicate detection and sequencing algorithm in TCP can be defeated, however, if a source TCP endpoint does not have any memory of the sequence numbers it last used on a given connection. For example, if the TCP implementation were to start all connections with sequence number 0, then upon the host rebooting, a TCP peer might reform 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 that 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 endpoint on a particular connection. Now suppose, at this instant, the host reboots 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 S1 may arrive and be treated as new packets by the receiver of the new incarnation of the connection.

The problem is that the recovering host may not know for how long it was down between rebooting 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 reboot - this is the "quiet time" specification. Hosts that prefer to avoid waiting and are willing to risk possible confusion of old and new packets at a given destination may choose not to wait for the "quiet time". Implementors may provide TCP users with the ability to select on a connection by connection basis whether to wait after a reboot, or may informally implement the "quiet 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 rebooting a block of space-time is occupied by the octets and SYN or FIN flags of any potentially still in-flight segments, and if a new connection is started too soon and uses any of the sequence numbers in the space-time footprint of those potentially still in-flight segments of the previous connection incarnation, there is a potential sequence number overlap area that could cause confusion at the receiver.

High performance cases will have shorter cycle times than those in the megabits per second that the base TCP design described above considers. At 1 Gbps, the cycle time is 34 seconds, only 3 seconds at 10 Gbps, and around a third of a second at 100 Gbps. In these higher performance cases, TCP Timestamp options and Protection Against Wrapped Sequences (PAWS) [48] provide the needed capability to detect and discard old duplicates.

3.5. Establishing a connection

The "three-way handshake" is the procedure used to establish a connection. This procedure normally is initiated by one TCP peer and responded to by another TCP peer. The procedure also works if two TCP peers simultaneously initiate the procedure. When simultaneous open occurs, each TCP peer receives a "SYN" segment that 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 endpoint doesn't deliver the data to the user until it is clear the data is valid (e.g., the data is buffered at the receiver until the connection reaches the ESTABLISHED state, given that the three-way handshake reduces the possibility of false connections). It is a trade-off between memory and messages to provide information for this checking.

The simplest 3WHS is shown in Figure 6. 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 peer A to TCP peer B, or arrival of a segment at B from A. Left arrows (<--), indicate the reverse. Ellipsis (...) indicates a segment that is still in the network (delayed). Comments appear in parentheses. TCP connection 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 Peer A		TCP Peer B
1. CLOSED		LISTEN
2. SYN-SENT	--> <SEQ=100><CTL=SYN>	--> SYN-RECEIVED
3. ESTABLISHED	<-- <SEQ=300><ACK=101><CTL=SYN,ACK>	<-- SYN-RECEIVED
4. ESTABLISHED	--> <SEQ=101><ACK=301><CTL=ACK>	--> ESTABLISHED
5. ESTABLISHED	--> <SEQ=101><ACK=301><CTL=ACK><DATA>	--> ESTABLISHED

Figure 6: Basic 3-Way Handshake for Connection Synchronization

In line 2 of Figure 6, TCP Peer A begins by sending a SYN segment indicating that it will use sequence numbers starting with sequence number 100. In line 3, TCP Peer B sends a SYN and acknowledges the SYN it received from TCP Peer A. Note that the acknowledgment field indicates TCP Peer B is now expecting to hear sequence 101, acknowledging the SYN that occupied sequence 100.

At line 4, TCP Peer A responds with an empty segment containing an ACK for TCP Peer B's SYN; and in line 5, TCP Peer 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 ACKs!).

Simultaneous initiation is only slightly more complex, as is shown in Figure 7. Each TCP peer's connection state cycles from CLOSED to SYN-SENT to SYN-RECEIVED to ESTABLISHED.

TCP Peer A		TCP Peer B
1. CLOSED		CLOSED
2. SYN-SENT	--> <SEQ=100><CTL=SYN>	...
3. SYN-RECEIVED	<-- <SEQ=300><CTL=SYN>	<-- SYN-SENT
4.	... <SEQ=100><CTL=SYN>	--> SYN-RECEIVED
5. SYN-RECEIVED	--> <SEQ=100><ACK=301><CTL=SYN,ACK>	...
6. ESTABLISHED	<-- <SEQ=300><ACK=101><CTL=SYN,ACK>	<-- SYN-RECEIVED
7.	... <SEQ=100><ACK=301><CTL=SYN,ACK>	--> ESTABLISHED

Figure 7: Simultaneous Connection Synchronization

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

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

The principal reason for the three-way handshake is to prevent old duplicate connection initiations from causing confusion. To deal with this, a special control message, reset, is specified. If the receiving TCP peer is in a non-synchronized state (i.e., SYN-SENT, SYN-RECEIVED), it returns to LISTEN on receiving an acceptable reset. If the TCP peer 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 Peer A		TCP Peer 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. ESTABLISHED	<-- <SEQ=400><ACK=101><CTL=SYN,ACK>	<-- SYN-RECEIVED
8. ESTABLISHED	--> <SEQ=101><ACK=401><CTL=ACK>	--> ESTABLISHED

Figure 8: Recovery from Old Duplicate SYN

As a simple example of recovery from old duplicates, consider Figure 8. At line 3, an old duplicate SYN arrives at TCP Peer B. TCP Peer B cannot tell that this is an old duplicate, so it responds normally (line 4). TCP Peer A detects that the ACK field is incorrect and returns a RST (reset) with its SEQ field selected to make the segment believable. TCP Peer B, on receiving the RST, returns to the LISTEN state. When the original SYN 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.

3.5.1. Half-Open Connections and Other Anomalies

An established connection is said to be "half-open" if one of the TCP peers 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 failure or reboot 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.

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 endpoint receiving a reset control message. Such a message indicates to the site B TCP endpoint 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 failure or reboot occurs causing loss of memory to A's TCP implementation. Depending on the operating system supporting A's TCP implementation, it is likely that some error recovery mechanism exists. When the TCP endpoint 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 implementation. In an attempt to establish the connection, A's TCP implementation will send a segment containing SYN. This scenario leads to the example shown in Figure 9. After TCP Peer A reboots, the user attempts to re-open the connection. TCP Peer B, in the meantime, thinks the connection is open.

TCP Peer A	TCP Peer B
1. (REBOOT)	(send 300, receive 100)
2. CLOSED	ESTABLISHED
3. SYN-SENT --> <SEQ=400><CTL=SYN>	--> (??)
4. (!!) <-- <SEQ=300><ACK=100><CTL=ACK>	<-- ESTABLISHED
5. SYN-SENT --> <SEQ=100><CTL=RST>	--> (Abort!!)
6. SYN-SENT	CLOSED
7. SYN-SENT --> <SEQ=400><CTL=SYN>	-->

Figure 9: Half-Open Connection Discovery

When the SYN arrives at line 3, TCP Peer 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 Peer 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 Peer B aborts at line 5. TCP Peer A will continue to try to establish the connection; the problem is now reduced to the basic 3-way handshake of Figure 6.

An interesting alternative case occurs when TCP Peer A reboots and TCP Peer B tries to send data on what it thinks is a synchronized connection. This is illustrated in Figure 10. In this case, the data arriving at TCP Peer A from TCP Peer B (line 2) is unacceptable because no such connection exists, so TCP Peer A sends a RST. The RST is acceptable so TCP Peer B processes it and aborts the connection.

TCP Peer A	TCP Peer B
1. (REBOOT)	(send 300, receive 100)
2. (??) <-- <SEQ=300><ACK=100><DATA=10><CTL=ACK>	<-- ESTABLISHED
3. --> <SEQ=100><CTL=RST>	--> (ABORT!!)

Figure 10: Active Side Causes Half-Open Connection Discovery

In Figure 11, two TCP Peers A and B with passive connections waiting for SYN are depicted. An old duplicate arriving at TCP Peer 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 Peer B accepts the reset and returns to its passive LISTEN state.

TCP Peer A	TCP Peer B
1. LISTEN	LISTEN
2. ... <SEQ=Z><CTL=SYN>	--> SYN-RECEIVED
3. (??) <-- <SEQ=X><ACK=Z+1><CTL=SYN,ACK>	<-- SYN-RECEIVED
4. --> <SEQ=Z+1><CTL=RST>	--> (return to LISTEN!)
5. LISTEN	LISTEN

Figure 11: Old Duplicate SYN Initiates a Reset on two Passive Sockets

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

3.5.2. Reset Generation

A TCP user or application can issue a reset on a connection at any time, though reset events are also generated by the protocol itself when various error conditions occur, as described below. The side of a connection issuing a reset should enter the TIME-WAIT state, as this generally helps to reduce the load on busy servers for reasons described in [71].

As a general rule, reset (RST) is sent whenever a segment arrives that 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. A SYN segment that does not match an existing connection is 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 Appendix A.1 that does not exactly match the level and compartment requested for the connection, a reset is sent.

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

3. If the connection is in a synchronized state (ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT), any unacceptable segment (out of window sequence number or unacceptable acknowledgment number) must be responded to with an empty acknowledgment segment (without any user data) 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 that does not exactly match the level and compartment requested for the connection, a reset is sent and the connection goes to the CLOSED state. The reset takes its sequence number from the ACK field of the incoming segment.

3.5.3. 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 implementations SHOULD allow a received RST segment to include data (SHLD-2). It has been suggested that a RST segment could contain diagnostic data that explains the cause of the RST. No standard has yet been established for such data.

3.6. 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 the TCP receiver is told that the remote peer 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 remote peer has CLOSED. The TCP implementation will signal a user, even if no RECEIVES are outstanding, that the remote peer has closed, so the user can terminate their side gracefully. A TCP implementation 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 their data was received at the destination TCP endpoint. Users must keep reading connections they close for sending until the TCP implementation indicates there is no more data.

There are essentially three cases:

- 1) The user initiates by telling the TCP implementation to CLOSE the connection (TCP Peer A in Figure 12).
- 2) The remote TCP endpoint initiates by sending a FIN control signal (TCP Peer B in Figure 12).
- 3) Both users CLOSE simultaneously (Figure 13).

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 implementation, 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 peer has both acknowledged the FIN and sent a FIN of its own, the first TCP peer can ACK this FIN. Note that a TCP endpoint receiving a FIN will ACK but not send its own FIN until its user has CLOSED the connection also.

Case 2: TCP endpoint receives a FIN from the network

If an unsolicited FIN arrives from the network, the receiving TCP endpoint can ACK it and tell the user that the connection is closing. The user will respond with a CLOSE, upon which the TCP endpoint can send a FIN to the other TCP peer after sending any remaining data. The TCP endpoint 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 (Figure 13). When all segments preceding the FINs have been processed and acknowledged, each TCP peer can ACK the FIN it has received. Both will, upon receiving these ACKs, delete the connection.

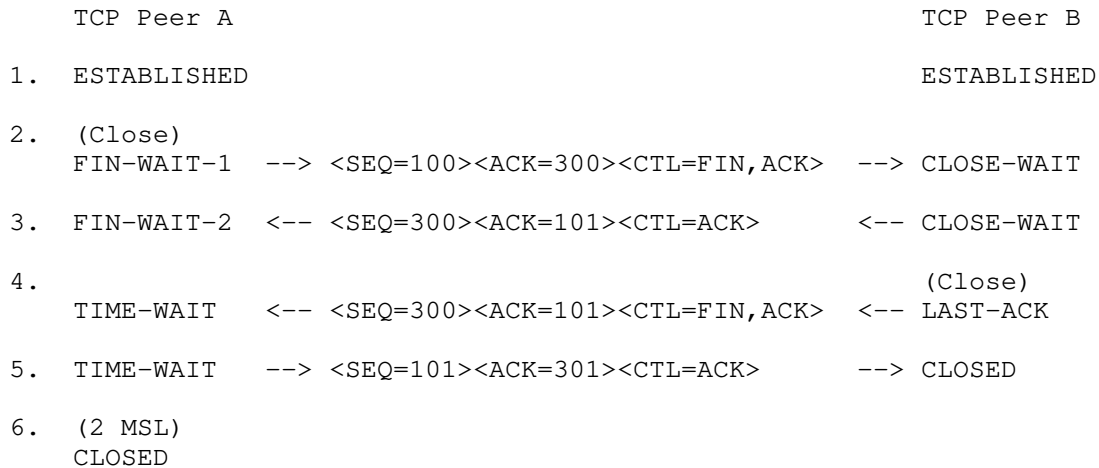


Figure 12: Normal Close Sequence

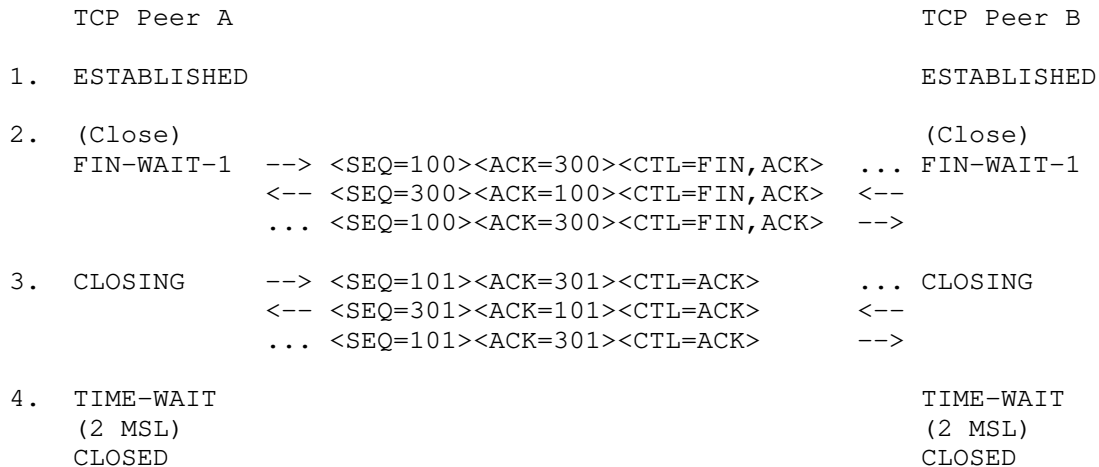


Figure 13: Simultaneous Close Sequence

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

3.6.1. Half-Closed Connections

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

A host MAY implement a "half-duplex" TCP close sequence, so that an application that has called CLOSE cannot continue to read data from the connection (MAY-1). If such a host issues a CLOSE call while received data is still pending in the TCP connection, or if new data is received after CLOSE is called, its TCP implementation SHOULD send a RST to show that data was lost (SHLD-3). See [24] section 2.17 for discussion.

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

- (1) assigns its initial sequence number for the new connection to be larger than the largest sequence number it used on the previous connection incarnation, and
- (2) returns to TIME-WAIT state if the SYN turns out to be an old duplicate.

When the TCP Timestamp options are available, an improved algorithm is described in [41] in order to support higher connection establishment rates. This algorithm for reducing TIME-WAIT is a Best Current Practice that SHOULD be implemented, since timestamp options are commonly used, and using them to reduce TIME-WAIT provides benefits for busy Internet servers (SHLD-4).

3.7. 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 Remote Direct Memory Access (RDMA) using Direct Data Placement (DDP) and Marker PDU

Aligned Framing (MPA) [35], 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:

- * Reducing the number of packets in flight within the network.
- * Increasing processing efficiency and potential performance by enabling a smaller number of interrupts and inter-layer interactions.
- * 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 implementation architectures, 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:

- * Avoiding sending a TCP segment that would result in an IP datagram larger than the smallest MTU along an IP network path, because this results in either packet loss or packet fragmentation. Making matters worse, some firewalls or middleboxes may drop fragmented packets or ICMP messages related to fragmentation.
- * 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.
- * 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 endpoints **MUST** implement both sending and receiving the MSS option (MUST-14).

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

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

The maximum size of a segment that TCP endpoint really sends, the "effective send MSS," **MUST** be the smaller (MUST-16) of the send MSS (that reflects the available reassembly buffer size at the remote host, the EMTU_R [20]) and the largest transmission size permitted by the IP layer (EMTU_S [20]):

$$\text{Eff.snd.MSS} =$$
$$\min(\text{SendMSS}+20, \text{MMS_S}) - \text{TCPhdrsize} - \text{IPOptionsize}$$

where:

- * 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.
- * MMS_S is the maximum size for a transport-layer message that TCP may send.
- * 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 might 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.
- * IPOptionsize is the size of any IPv4 options or IPv6 extension headers associated with a TCP connection. Note that some options or extension headers might 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 [44] discusses this in greater detail.

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

$MMS_R - 20$

where MMS_R is the maximum size for a transport-layer message that can be received (and reassembled at the IP layer) (MUST-67). 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 $EMTU_S$ [20].

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 recommends an IP-layer default effective MTU of less than or equal to 576 for destinations not directly connected, and for IPv6 this would be 1280. Using these fixed values limits TCP connection performance and efficiency. Instead, 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 [14] 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 [28]. PLPMTUD [32] 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 (see [44]). 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 the TCP MSS to 536 (corresponding to the IPv4 576 byte default MTU) 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) [38]. It is tempting for a TCP implementation 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 packet-to-packet, TCP implementations SHOULD use the smallest effective MTU of the interface to calculate the value to advertise in the MSS option (SHLD-6).

3.7.4. Nagle Algorithm

The "Nagle algorithm" was described in RFC 896 [18] and was recommended in RFC 1122 [20] for mitigation of an early problem of too many small packets being generated. It has been implemented in most current TCP code bases, sometimes with minor variations (see Appendix A.3).

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

A TCP implementation SHOULD implement the Nagle Algorithm to coalesce short segments (SHLD-7). However, there MUST be a way for an application to disable the Nagle algorithm on an individual connection (MUST-17). In all cases, sending data is also subject to the limitation imposed by the Slow Start algorithm [8].

Since there can be problematic interactions between the Nagle Algorithm and delayed acknowledgements, some implementations use minor variations of the Nagle algorithm, such as the one described in Appendix A.3.

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 [25] defines that an MSS value of 65,535 bytes is to be treated as infinity, and Path MTU Discovery [14] is used to determine the actual MSS.

The Jumbo Payload option need not be implemented or understood by IPv6 nodes that do not support attachment to links with a MTU greater than 65,575 [25], and the present IPv6 Node Requirements does not include support for Jumbograms [55].

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 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 implementation 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 (see Section 3.9.1), 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 (MUST-18).

RFC 793 contains an early example procedure for computing the RTO, based on work mentioned in IEN 177 [72]. This was then replaced by the algorithm described in RFC 1122, and subsequently updated in RFC 2988, and then again in RFC 6298.

RFC 1122 allows that if a retransmitted packet is identical to the original packet (which implies not only that the data boundaries have not changed, but also that none of the headers have changed), then the same IPv4 Identification field MAY be used (see Section 3.2.1.5 of RFC 1122) (MAY-4). The same IP identification field may be reused anyways, since it is only meaningful when a datagram is fragmented [45]. TCP implementations should not rely on or typically interact with this IPv4 header field in any way. It is not a reasonable way to either indicate duplicate sent segments, nor to identify duplicate received segments.

3.8.2. TCP Congestion Control

RFC 2914 [5] explains the importance of congestion control for the Internet.

RFC 1122 required implementation of Van Jacobson's congestion control algorithms slow start and congestion avoidance together with exponential back-off for successive RTO values for the same segment. RFC 2581 provided IETF Standards Track description of slow start and congestion avoidance, along with fast retransmit and fast recovery. RFC 5681 is the current description of these algorithms and is the current Standards Track specification providing guidelines for TCP congestion control. RFC 6298 describes exponential back-off of RTO values, including keeping the backed-off value until a subsequent segment with new data has been sent and acknowledged without retransmission.

A TCP endpoint MUST implement the basic congestion control algorithms slow start, congestion avoidance, and exponential back-off of RTO to avoid creating congestion collapse conditions (MUST-19). RFC 5681 and RFC 6298 describe the basic algorithms on the IETF Standards Track that are broadly applicable. Multiple other suitable algorithms exist and have been widely used. Many TCP implementations

support a set of alternative algorithms that can be configured for use on the endpoint. An endpoint MAY implement such alternative algorithms provided that the algorithms are conformant with the TCP specifications from the IETF Standards Track as described in RFC 2914, RFC 5033 [7], and RFC 8961 [15] (MAY-18).

Explicit Congestion Notification (ECN) was defined in RFC 3168 and is an IETF Standards Track enhancement that has many benefits [52].

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

3.8.3. TCP Connection Failures

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

- (a) There are two thresholds R1 and R2 measuring the amount of retransmission that has occurred for the same segment. R1 and R2 might be measured in time units or as a count of retransmissions (with the current RTO and corresponding backoffs as a conversion factor, if needed).
- (b) When the number of transmissions of the same segment reaches or exceeds threshold R1, pass negative advice (see Section 3.3.1.4 of [20]) to the IP layer, to trigger dead-gateway diagnosis.
- (c) When the number of transmissions of the same segment reaches a threshold R2 greater than R1, close the connection.
- (d) An application MUST (MUST-21) be able to set the value for R2 for a particular connection. For example, an interactive application might set R2 to "infinity," giving the user control over when to disconnect.
- (e) TCP implementations SHOULD inform the application of the delivery problem (unless such information has been disabled by the application; see Asynchronous Reports section), when R1 is reached and before R2 (SHLD-9). This will allow a remote login application program to inform the user, for example.

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

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

However, the values of R1 and R2 may be different for SYN and data segments. In particular, R2 for a SYN segment MUST be set large enough to provide retransmission of the segment for at least 3 minutes (MUST-23). The application can close the connection (i.e., give up on the open attempt) sooner, of course.

3.8.4. TCP Keep-Alives

A TCP connection is said to be "idle" if for some long amount of time there have been no incoming segments received and there is no new or unacknowledged data to be sent.

Implementors MAY include "keep-alives" in their TCP implementations (MAY-5), although this practice is not universally accepted. Some TCP implementations, however, have included a keep-alive mechanism. To confirm that an idle connection is still active, these implementations send a probe segment designed to elicit a response from the TCP peer. Such a segment generally contains SEG.SEQ = SND.NXT-1 and may or may not contain one garbage octet of data. If keep-alives are included, the application MUST be able to turn them on or off for each TCP connection (MUST-24), and they MUST default to off (MUST-25).

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

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

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

3.8.5. The Communication of Urgent Information

As a result of implementation differences and middlebox interactions, new applications SHOULD NOT employ the TCP urgent mechanism (SHLD-13). However, TCP implementations MUST still include support for the urgent mechanism (MUST-30). Information on how some TCP implementations interpret the urgent pointer can be found in RFC 6093 [40].

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 endpoint 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 endpoint, that TCP must tell the user to go into "urgent mode"; when the receive sequence number catches up to the urgent pointer, the TCP implementation 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 an urgent field that 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. Note that because changes in the urgent pointer correspond to data being written by a sending application, the urgent pointer can not "recede" in the sequence space, but a TCP receiver should be robust to invalid urgent pointer values.

A TCP implementation MUST support a sequence of urgent data of any length (MUST-31). [20]

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

A TCP implementation MUST (MUST-32) 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. The TCP implementation MUST (MUST-33)

provide a way for the application to learn how much urgent data remains to be read from the connection, or at least to determine whether more urgent data remains to be read [20].

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 endpoint packages the data to be transmitted into segments that 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 TCP endpoints. 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 endpoint 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 [20] dictates that TCP peers will not shrink the window themselves, but will be prepared for such behavior on the part of other TCP peers.

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

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

3.8.6.1. Zero Window Probing

The sending TCP peer must regularly transmit at least one octet of new data (if available) or retransmit to the receiving TCP peer even if the send window is zero, in order to "probe" the window. This retransmission is essential to guarantee that when either TCP peer has a zero window the re-opening of the window will be reliably reported to the other. This is referred to as Zero-Window Probing (ZWP) in other documents.

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

A TCP implementation MAY keep its offered receive window closed indefinitely (MAY-8). As long as the receiving TCP peer continues to send acknowledgments in response to the probe segments, the sending TCP peer MUST allow the connection to stay open (MUST-37). This enables TCP to function in scenarios such as the "printer ran out of paper" situation described in Section 4.2.2.17 of [20]. The behavior is subject to the implementation's resource management concerns, as noted in [42].

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

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

3.8.6.2. Silly Window Syndrome Avoidance

The "Silly Window Syndrome" (SWS) is a stable pattern of small incremental window movements resulting in extremely poor TCP performance. Algorithms to avoid SWS are described below for both the sending side and the receiving side. RFC 1122 contains more detailed discussion of the SWS problem. Note that the Nagle algorithm and the sender SWS avoidance algorithm play complementary roles in improving performance. The Nagle algorithm discourages

sending tiny segments when the data to be sent increases in small increments, while the SWS avoidance algorithm discourages small segments resulting from the right window edge advancing in small increments.

3.8.6.2.1. Sender's Algorithm - When to Send Data

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

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

The sender's SWS avoidance algorithm is more difficult than the receiver's, because the sender does not know (directly) the receiver's total buffer space RCV.BUFF. An approach that 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 "usable 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 endpoint 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:

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

(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 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 3.8.6.1).

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

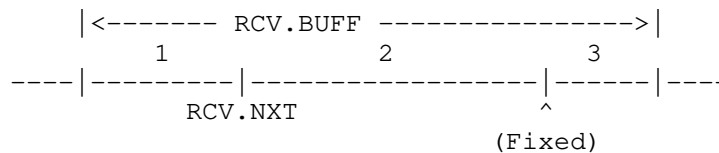
A TCP implementation MUST include a SWS avoidance algorithm in the receiver (MUST-39).

The receiver's SWS avoidance algorithm determines when the right window edge may be advanced; this is customarily known as "updating the window". This algorithm combines with the delayed ACK algorithm (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 that 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:

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

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, making the assumption that 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 endpoint SHOULD implement a delayed ACK (SHLD-18), but an ACK should not be excessively delayed; in particular, the delay MUST be less than 0.5 seconds (MUST-40). An ACK SHOULD be generated for at least every second full-sized segment or 2*RMSS bytes of new data (where RMSS is the MSS specified by the TCP endpoint receiving the segments to be acknowledged, or the default value if not specified) (SHLD-19). Excessive delays on ACKs can disturb the round-trip timing and packet "clocking" algorithms. More complete discussion of delayed ACK behavior is in Section 4.2 of RFC 5681 [8], including recommendations to immediately acknowledge out-of-order segments, segments above a gap in sequence space, or segments that fill all or part of a gap, in order to accelerate loss recovery.

Note that there are several current practices that further lead to a reduced number of ACKs, including generic receive offload (GRO) [73], ACK compression, and ACK decimation [29].

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 TCP implementations might use.

3.9.1. User/TCP Interface

The following functional description of user commands to the TCP implementation 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 TCP implementations 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.

Section 3.1 of [54] also identifies primitives provided by TCP, and could be used as an additional reference for implementers.

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

The user commands described below specify the basic functions the TCP implementation 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 implementation 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 remote socket).
- (b) replies to specific user commands indicating success or various types of failure.

3.9.1.1. Open

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

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 remote socket to wait for a particular connection or an unspecified remote socket to wait for any call. A fully specified passive call can be made active by the subsequent execution of a SEND.

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

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

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

On an active OPEN command, the TCP endpoint 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 endpoint will abort the connection. The present global default is five minutes.

The TCP implementation or some component of the operating system will verify the user's authority to open a connection with the specified DiffServ field value or security/compartment. The absence of a DiffServ field value or security/compartment specification in the OPEN call indicates the default values must be used.

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

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

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

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

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

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

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

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

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

3.9.1.2. Send

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

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

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

If the PUSH flag is set, the application intends the data to be transmitted promptly to the receiver, and the PUSH bit will be set in the last TCP segment created from the buffer.

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

If the PUSH flag is not set, the data may be combined with data from subsequent SENDs for transmission efficiency. When an application issues a series of SEND calls without setting the PUSH flag, the TCP implementation MAY aggregate the data internally without sending it (MAY-16). Note that when the Nagle algorithm is in use, TCP implementations may buffer the data before sending, without regard to the PUSH flag (see Section 3.7.4).

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

New applications SHOULD NOT set the URGENT flag [40] due to implementation differences and middlebox issues (SHLD-13).

If the URGENT flag is set, segments sent to the destination TCP peer will have the urgent pointer set. The receiving TCP peer 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 implementation 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 remote socket was specified in the OPEN, but the connection is established (e.g., because a LISTENing connection has become specific due to a remote segment arriving for the local socket), then the designated buffer is sent to the implied remote socket. Users who make use of OPEN with an unspecified remote socket can make use of SEND without ever explicitly knowing the remote socket address.

However, if a SEND is attempted before the remote socket becomes specified, an error will be returned. Users can use the STATUS call to determine the status of the connection. Some TCP implementations 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 endpoint 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 endpoint. 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 endpoint. 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 that should be returned to the calling process.

3.9.1.3. Receive

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

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

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

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

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

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

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

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

3.9.1.4. 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 remote peer 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 peer gives up.

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

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

Close also implies push function.

3.9.1.5. 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,
- remote socket,
- local connection name,
- receive window,
- send window,
- connection state,
- number of buffers awaiting acknowledgment,
- number of buffers pending receipt,
- urgent state,
- DiffServ field value,
- security/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.

3.9.1.6. 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 remote TCP peer 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.

3.9.1.7. Flush

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

3.9.1.8. Asynchronous Reports

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

- ERROR_REPORT(local connection name, reason, subreason)

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

- * ICMP error message arrived (see Section 3.9.2.2 for description of handling each ICMP message type, since some message types need to be suppressed from generating reports to the application)
- * Excessive retransmissions (see Section 3.8.3)
- * Urgent pointer advance (see Section 3.8.5)

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

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

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

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

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

3.9.2. TCP/Lower-Level Interface

The TCP endpoint 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 [13].

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

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

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

- Note that RFC 793 specified one minute (60 seconds) as a constant for the TTL, because the assumed maximum segment lifetime was two minutes. This was intended to explicitly ask that a segment be destroyed if it cannot be delivered by the internet system within one minute. RFC 1122 changed this specification to require that the TTL be configurable.
- Note that the DiffServ field is permitted to change during a connection (Section 4.2.4.2 of RFC 1122). However, the application interface might not support this ability, and the application does not have knowledge about individual TCP segments, so this can only be done on a coarse granularity, at best. This limitation is further discussed in RFC 7657 (sec 5.1, 5.3, and 6) [51]. Generally, an application SHOULD NOT change the DiffServ field value during the course of a connection (SHLD-23).

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

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

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

3.9.2.1. Source Routing

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

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

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

3.9.2.2. ICMP Messages

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

This applies to ICMPv6 in addition to IPv4 ICMP.

[36] 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 implementations MUST silently discard any received ICMP Source Quench messages (MUST-55). See [11] for discussion.

Soft Errors

For IPv4 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, 3; Time Exceeded -- codes 0, 1; and Parameter Problem -- codes 0, 1, 2.

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

Hard Errors

For ICMP these include Destination Unreachable -- codes 2-4.

These are hard error conditions, so TCP implementations **SHOULD** abort the connection (SHLD-26). [36] 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 [36] section 4 describes widespread implementation behavior that treats soft errors as hard errors during connection establishment.

3.9.2.3. Source Address Validation

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

An incoming SYN with an invalid source address **MUST** be ignored either by TCP or by the IP layer (MUST-63) (Section 3.2.1.3 of [20]).

A TCP implementation **MUST** silently discard an incoming SYN segment that is addressed to a broadcast or multicast address (MUST-57).

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

3.10. Event Processing

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

The activity of the TCP endpoint 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 endpoint 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 in this document are identified by 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 connection stays in the same state.

3.10.1. 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, remote socket, DiffServ field, security/compartments, and user timeout information. Note that some parts of the remote socket may be unspecified in a passive OPEN and are to be filled in by the parameters of the incoming SYN segment. Verify the security and DiffServ value requested are allowed for this user, if not return "error: DiffServ value not allowed" or "error: security/compartments not allowed." If passive enter the LISTEN state and return. If active and the remote socket is unspecified, return "error: remote socket unspecified"; if active and the remote 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 the OPEN call is active and the remote 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 the remote socket was not specified, then return "error: remote 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".

3.10.2. 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 remote 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 the remote socket was not specified, then return "error: remote socket unspecified".

SYN-SENT STATE

SYN-RECEIVED STATE

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

ESTABLISHED STATE

CLOSE-WAIT STATE

- Segmentize the buffer and send it with a piggybacked acknowledgment (acknowledgment value = RCV.NXT). If there is insufficient space to remember this buffer, simply return "error: insufficient resources".
- If the urgent flag is set, then SND.UP <- SND.NXT and set the urgent pointer in the outgoing segments.

FIN-WAIT-1 STATE

FIN-WAIT-2 STATE

CLOSING STATE

LAST-ACK STATE

TIME-WAIT STATE

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

3.10.3. 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 endpoint 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 data already on hand, but not yet delivered to the user. If no text is awaiting delivery, the RECEIVE will get an "error: connection closing" response. Otherwise, any remaining data can be used to satisfy the RECEIVE.

CLOSING STATE

LAST-ACK STATE

TIME-WAIT STATE

- Return "error: connection closing".

3.10.4. 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 an "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".

3.10.5. 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:
 - o <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.

3.10.6. 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.

3.10.7. SEGMENT ARRIVES

3.10.7.1. CLOSED State

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 endpoint 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.

3.10.7.2. LISTEN State

If the state is LISTEN then

first check for an RST

- 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. 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:

- o <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.
 - o <SEQ=0><ACK=SEG.SEQ+SEG.LEN><CTL=RST,ACK>
- Set RCV.NXT to SEG.SEQ+1, IRS is set to SEG.SEQ and any other control or text should be queued for processing later. ISS should be selected and a SYN segment sent of the form:
 - o <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 remote socket was not fully specified), then the unspecified fields should be filled in now.

fourth other data or control

- This should not be reached. Drop the segment and return. Any other control or data-bearing segment (not containing SYN) must have an ACK and thus would have been discarded by the ACK processing in the second step, unless it was first discarded by RST checking in the first step.

3.10.7.3. SYN-SENT State

If the state is SYN-SENT then

first check the ACK bit

- If the ACK bit is set
 - o 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>
 - o and discard the segment. Return.

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

second check the RST bit

- If the RST bit is set
 - o A potential blind reset attack is described in RFC 5961 [9]. The mitigation described in that document has specific applicability explained therein, and is not a substitute for cryptographic protection (e.g. IPsec or TCP-AO). A TCP implementation that supports the RFC 5961 mitigation SHOULD first check that the sequence number exactly matches `RCV.NXT` prior to executing the action in the next paragraph.
 - o If the ACK was acceptable then signal the user "error: connection reset", drop the segment, enter CLOSED state, delete TCB, and return. Otherwise (no ACK), drop the segment and return.

third check the security

- If the security/compartments in the segment does not exactly match the security/compartments in the TCB, send a reset
 - o If there is an ACK
 - + `<SEQ=SEG.ACK><CTL=RST>`
 - o Otherwise
 - + `<SEQ=0><ACK=SEG.SEQ+SEG.LEN><CTL=RST,ACK>`
- If a reset was sent, discard the segment and return.

fourth check the SYN bit

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

- If the SYN bit is on and the security/compartiment is acceptable then, RCV.NXT is set to SEG.SEQ+1, IRS is set to SEG.SEQ. SND.UNA should be advanced to equal SEG.ACK (if there is an ACK), and any segments on the retransmission queue that 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
 - o <SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>
 - and send it. Data or controls that were queued for transmission MAY be included. Some TCP implementations suppress sending this segment when the received segment contains data that will anyways generate an acknowledgement in the later processing steps, saving this extra acknowledgement of the SYN from being sent. If there are other controls or text in the segment then continue processing at the sixth step under Section 3.10.7.4 where the URG bit is checked, otherwise return.
 - Otherwise enter SYN-RECEIVED, form a SYN,ACK segment
 - o <SEQ=ISS><ACK=RCV.NXT><CTL=SYN,ACK>
 - and send it. Set the variables:
 - o SND.WND <- SEG.WND
 - SND.WL1 <- SEG.SEQ
 - SND.WL2 <- SEG.ACK
- If there are other controls or text in the segment, queue them for processing after the ESTABLISHED state has been reached, return.
- Note that it is legal to send and receive application data on SYN segments (this is the "text in the segment" mentioned above. There has been significant misinformation and misunderstanding of this topic historically. Some firewalls and security devices consider this suspicious. However, the capability was used in T/TCP [22] and is used in TCP Fast Open (TFO) [49], so is important for implementations and network devices to permit.

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

3.10.7.4. Other States

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

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

- o In implementing sequence number validation as described here, please note Appendix A.2.
- o If the RCV.WND is zero, no segments will be acceptable, but special allowance should be made to accept valid ACKs, URGs and RSTs.
- o 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>
- o After sending the acknowledgment, drop the unacceptable segment and return.
- o Note that for the TIME-WAIT state, there is an improved algorithm described in [41] for handling incoming SYN segments, that utilizes timestamps rather than relying on the sequence number check described here. When the improved algorithm is implemented, the logic above is not applicable for incoming SYN segments with timestamp options, received on a connection in the TIME-WAIT state.
- o In the following it is assumed that the segment is the idealized segment that begins at RCV.NXT and does not exceed the window. One could tailor actual segments to fit this assumption by trimming off any portions that lie outside the window (including SYN and FIN), and only processing further if the segment then begins at RCV.NXT. Segments with higher beginning sequence numbers SHOULD be held for later processing (SHLD-31).
- second check the RST bit,

- o RFC 5961 [9] section 3 describes a potential blind reset attack and optional mitigation approach. This does not provide a cryptographic protection (e.g. as in IPsec or TCP-AO), but can be applicable in situations described in RFC 5961. For stacks implementing the RFC 5961 protection, 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 endpoints 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 endpoints MUST send an acknowledgement (challenge ACK):

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

After sending the challenge ACK, TCP endpoints 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.

- o 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, the retransmission queue should be flushed. And in the active OPEN case, enter the CLOSED state and delete the TCB, and return.

- o 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.

o 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

o SYN-RECEIVED

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

o ESTABLISHED

FIN-WAIT-1

FIN-WAIT-2

CLOSE-WAIT

CLOSING

LAST-ACK

TIME-WAIT

- + If the security/compartments in the segment does not exactly match the security/compartments 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.

- o Note this check is placed following the sequence check to prevent a segment from an old connection between these port numbers with a different security from causing an abort of the current connection.
- fourth, check the SYN bit,
 - o SYN-RECEIVED
 - + If the connection was initiated with a passive OPEN, then return this connection to the LISTEN state and return. Otherwise, handle per the directions for synchronized states below.

ESTABLISHED STATE

FIN-WAIT STATE-1

FIN-WAIT STATE-2

CLOSE-WAIT STATE

CLOSING STATE

LAST-ACK STATE

TIME-WAIT STATE

- + If the SYN bit is set in these synchronized states, it may be either a legitimate new connection attempt (e.g. in the case of TIME-WAIT), an error where the connection should be reset, or the result of an attack attempt, as described in RFC 5961 [9]. For the TIME-WAIT state, new connections can be accepted if the timestamp option is used and meets expectations (per [41]). For all other cases, RFC 5961 provides a mitigation with applicability to some situations, though there are also alternatives that offer cryptographic protection (see Section 7). RFC 5961 recommends that in these synchronized states, if the SYN bit is set, irrespective of the sequence number, TCP endpoints MUST send a "challenge ACK" to the remote peer:
 - + <SEQ=SEND.NXT><ACK=RCV.NXT><CTL=ACK>
- + After sending the acknowledgement, TCP implementations 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,
 - o if the ACK bit is off drop the segment and return
 - o if the ACK bit is on
- + RFC 5961 [9] section 5 describes a potential blind data injection attack, and mitigation that implementations MAY choose to include (MAY-12). TCP stacks that implement RFC 5961 MUST add an input check that the ACK value is acceptable only if it is in the range of ((SND.UNA - MAX.SND.WND) =< SEG.ACK =< SND.NXT). All incoming segments whose ACK value doesn't satisfy the above condition MUST be discarded and an ACK sent back. The new state variable MAX.SND.WND is defined as the largest window that the local sender has ever received from its peer (subject to window scaling) or may be hard-coded to a maximum permissible window value. When the ACK value is acceptable, the processing per-state below applies:
- + SYN-RECEIVED STATE
 - * If $\text{SND.UNA} < \text{SEG.ACK} \leq \text{SND.NXT}$ then enter ESTABLISHED state and continue processing with variables below set to:
 - $\text{SND.WND} \leftarrow \text{SEG.WND}$
 - $\text{SND.WL1} \leftarrow \text{SEG.SEQ}$
 - $\text{SND.WL2} \leftarrow \text{SEG.ACK}$
 - * If the segment acknowledgment is not acceptable, form a reset segment,

- <SEQ=SEG.ACK><CTL=RST>
- * and send it.
- + ESTABLISHED STATE
 - * If `SND.UNA < SEG.ACK =< SND.NXT` then, set `SND.UNA <- SEG.ACK`. Any segments on the retransmission queue that are thereby entirely acknowledged are removed. Users should receive positive acknowledgments for buffers that have been SENT and fully acknowledged (i.e., SEND buffer should be returned with "ok" response). If the ACK is a duplicate (`SEG.ACK =< SND.UNA`), it can be ignored. If the ACK acks something not yet sent (`SEG.ACK > SND.NXT`) then send an ACK, drop the segment, and return.
 - * If `SND.UNA =< SEG.ACK =< SND.NXT`, the send window should be updated. If (`SND.WL1 < SEG.SEQ` or (`SND.WL1 = SEG.SEQ` and `SND.WL2 =< SEG.ACK`)), set `SND.WND <- SEG.WND`, set `SND.WL1 <- SEG.SEQ`, and set `SND.WL2 <- SEG.ACK`.
 - * Note that `SND.WND` is an offset from `SND.UNA`, that `SND.WL1` records the sequence number of the last segment used to update `SND.WND`, and that `SND.WL2` records the acknowledgment number of the last segment used to update `SND.WND`. The check here prevents using old segments to update the window.
- + FIN-WAIT-1 STATE
 - * In addition to the processing for the ESTABLISHED state, if the FIN segment 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,
 - o 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.
 - o 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,
 - o ESTABLISHED STATE

FIN-WAIT-1 STATE

FIN-WAIT-2 STATE

- + Once in the ESTABLISHED state, it is possible to deliver segment data to user RECEIVE buffers. Data from segments can be moved into buffers until either the buffer is full or the segment is empty. If the segment empties and carries a PUSH flag, then the user is informed, when the buffer is returned, that a PUSH has been received.
- + When the TCP endpoint takes responsibility for delivering the data to the user it must also acknowledge the receipt of the data.
- + Once the TCP endpoint 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 implementation MAY send an ACK segment acknowledging RCV.NXT when a valid segment arrives that is in the window but not at the left window edge (MAY-13).
- + Please note the window management suggestions in Section 3.8.
- + Send an acknowledgment of the form:
 - * <SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>
- + This acknowledgment should be piggybacked on a segment being transmitted if possible without incurring undue delay.

o 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,
 - o 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.
 - o 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.

3.10.8. Timeouts

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.

4. Glossary

ACK

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

connection

A logical communication path identified by a pair of sockets.

datagram

A message sent in a packet switched computer communications network.

Destination Address

The network layer address of the endpoint intended to receive a segment.

FIN

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

- flush**
To remove all of the contents (data or segments) from a store (buffer or queue).
- fragment**
A portion of a logical unit of data, in particular an internet fragment is a portion of an internet datagram.
- header**
Control information at the beginning of a message, segment, fragment, packet or block of data.
- host**
A computer. In particular a source or destination of messages from the point of view of the communication network.
- Identification**
An Internet Protocol field. This identifying value assigned by the sender aids in assembling the fragments of a datagram.
- internet address**
A network layer address.
- internet datagram**
A unit of data exchanged between internet hosts, together with the internet header that allows the datagram to be routed from source to destination.
- internet fragment**
A portion of the data of an internet datagram with an internet header.
- IP**
Internet Protocol. See [1] and [13].
- IRS**
The Initial Receive Sequence number. The first sequence number used by the sender on a connection.
- ISN**
The Initial Sequence Number. The first sequence number used on a connection, (either ISS or IRS). Selected in a way that is unique within a given period of time and is unpredictable to attackers.
- ISS**
The Initial Send Sequence number. The first sequence number used by the sender on a connection.

left sequence

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

module

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

MSL

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

octet

An eight bit byte.

Options

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

packet

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

port

The portion of a connection identifier used for demultiplexing connections at an endpoint.

process

A program in execution. A source or destination of data from the point of view of the TCP endpoint 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 endpoint is expecting to receive.

receive window

This represents the sequence numbers the local (receiving) TCP endpoint is willing to receive. Thus, the local TCP endpoint 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 this range are considered duplicates or injection attacks and discarded.

RST

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

SEG.ACK

segment acknowledgment

SEG.LEN

segment length

SEG.SEQ

segment sequence

SEG.UP

segment urgent pointer field

SEG.WND

segment window field

segment

A logical unit of data, in particular a TCP segment is the unit of data 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 that 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 endpoint 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 that the remote (receiving) TCP endpoint is willing to receive. It is the value of the window field specified in segments from the remote (data receiving) TCP endpoint. The range of new sequence numbers that may be emitted by a TCP implementation lies between `SND.NXT` and `SND.UNA + SND.WND - 1`. (Retransmissions of sequence numbers between `SND.UNA` and `SND.NXT` are expected, of course.)

`SND.NXT`

send sequence

`SND.UNA`

left sequence

`SND.UP`

send urgent pointer

`SND.WL1`

segment sequence number at last window update

`SND.WL2`

segment acknowledgment number at last window update

`SND.WND`

send window

socket (or socket number, or socket address, or socket identifier)

An address that specifically includes a port identifier, that is, the concatenation of an Internet Address with a TCP port.

Source Address

The network layer address of the sending endpoint.

SYN

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

TCB

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

TCP

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

TOS

Type of Service, an obsoleted IPv4 field. The same header bits currently are used for the Differentiated Services field [4] containing the Differentiated Services Code Point (DSCP) value and the 2-bit ECN codepoint [6].

Type of Service

See "TOS".

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 by 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 that indicates the data octet associated with the sending user's urgent call.

5. Changes from RFC 793

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

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

The collection of applicable RFC Errata that have been reported and either accepted or held for an update to RFC 793 were incorporated (Errata IDs: 573, 574, 700, 701, 1283, 1561, 1562, 1564, 1571, 1572, 2297, 2298, 2748, 2749, 2934, 3213, 3300, 3301, 6222). Some errata were not applicable due to other changes (Errata IDs: 572, 575, 1565, 1569, 2296, 3305, 3602).

Changes to the specification of the Urgent Pointer described in RFCs 1011, 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 1011 [19] contains a number of comments about RFC 793, including some needed changes to the TCP specification. These are expanded in RFC 1122, which 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. The present document updates RFC 1011, since this is now the TCP specification rather than RFC 793, and the comments noted in 1011 have been incorporated.

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.

The description of congestion control implementation was added, based on the set of documents that are IETF BCP or Standards Track on the topic, and the current state of common implementations.

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 report basically is just a reminder that RFC 1122 updates 793. Some of the associated changes are left pending to a separate revision that incorporates 1122. Bob's mention of PUSH in 793 section 2.8 was not applicable here because that section was not part of the "functional specification". Also, the 1122 text on the retransmission timeout also has been updated by subsequent RFCs, so the change here deviates from Bob's suggestion to apply the 1122 text.)

Errata ID 574: Reported by Yin Shuming

Errata ID 700: Reported by Yin Shuming

Errata ID 701: Reported by Yin Shuming

Errata ID 1283: Reported by Pei-chun Cheng

Errata ID 1561: Reported by Constantin Hagemeier

Errata ID 1562: Reported by Constantin Hagemeier

Errata ID 1564: Reported by Constantin Hagemeier

Errata ID 1565: Reported by Constantin Hagemeier

Errata ID 1571: Reported by Constantin Hagemeier

Errata ID 1572: Reported by Constantin Hagemeier

Errata ID 2296: Reported by Vishwas Manral

Errata ID 2297: Reported by Vishwas Manral

Errata ID 2298: Reported by Vishwas Manral

Errata ID 2748: Reported by Mykyta Yevstifeyev

Errata ID 2749: Reported by Mykyta Yevstifeyev

Errata ID 2934: Reported by Constantin Hagemeier

Errata ID 3213: Reported by EugnJun Yi

Errata ID 3300: Reported by Botong Huang

Errata ID 3301: Reported by Botong Huang

Errata ID 3305: Reported by Botong Huang

Note: Some verified errata were not used in this update, as they relate to sections of RFC 793 elided from this document. These include Errata ID 572, 575, and 1569.

Note: Errata ID 3602 was not applied in this revision as it is duplicative of the 1122 corrections.

Not related to RFC 793 content, this revision also makes small tweaks to the introductory text, fixes indentation of the pseudo header 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 [17], 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 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.

The -07 revision includes reference to RFC 6191, updated security considerations, discussion of additional implementation considerations, and clarification of data on the SYN.

The -08 revision includes changes based on:

- describing treatment of reserved bits (following TCPM mailing list thread from July 2014 on "793bis item - reserved bit behavior"
- addition a brief TCP key concepts section to make up for not including the outdated section 2 of RFC 793
- changed "TCP" to "host" to resolve conflict between 1122 wording on whether TCP or the network layer chooses an address when multihomed
- fixed/updated definition of options in glossary
- moved note on aggregating ACKs from 1122 to a more appropriate location
- resolved notes on IP precedence and security/compartments

added implementation note on sequence number validation
added note that PUSH does not apply when Nagle is active
added 1122 content on asynchronous reports to replace 793 section
on TCP to user messages

The -09 revision fixes section numbering problems.

The -10 revision includes additions to the security considerations based on comments from Joe Touch, and suggested edits on RST/FIN notification, RFC 2525 reference, and other edits suggested by Yuchung Cheng, as well as modifications to DiffServ text from Yuchung Cheng and Gorry Fairhurst.

The -11 revision includes a start at identifying all of the requirements text and referencing each instance in the common table at the end of the document.

The -12 revision completes the requirement language indexing started in -11 and adds necessary description of the PUSH functionality that was missing.

The -13 revision contains only changes in the inline editor notes.

The -14 revision includes updates with regard to several comments from the mailing list, including editorial fixes, adding IANA considerations for the header flags, improving figure title placement, and breaking up the "Terminology" section into more appropriately titled subsections.

The -15 revision has many technical and editorial corrections from Gorry Fairhurst's review, and subsequent discussion on the TCPM list, as well as some other collected clarifications and improvements from mailing list discussion.

The -16 revision addresses several discussions that rose from additional reviews and follow-up on some of Gorry Fairhurst's comments from revision 14.

The -17 revision includes errata 6222 from Charles Deng, update to the key words boilerplate, updated description of the header flags registry changes, and clarification about connections rather than users in the discussion of OPEN calls.

The -18 revision includes editorial changes to the IANA considerations, based on comments from Richard Scheffenegger at the IETF 108 TCPM virtual meeting.

The -19 revision includes editorial changes from Errata 6281 and 6282 reported by Merlin Buge. It also includes WGLC changes noted by Mohamed Boucadair, Rahul Jadhav, Praveen Balasubramanian, Matt Olson, Yi Huang, Joe Touch, and Juhamatti Kuusisaari.

The -20 revision includes text on congestion control based on mailing list and meeting discussion, put together in its final form by Markku Kojo. It also clarifies that SACK, WS, and TS options are recommended for high performance, but not needed for basic interoperability. It also clarifies that the length field is required for new TCP options.

The -21 revision includes slight changes to the header diagram for compatibility with tooling, from Stephen McQuistin, clarification on the meaning of idle connections from Yuchung Cheng, Neal Cardwell, Michael Scharf, and Richard Scheffenegger, editorial improvements from Markku Kojo, notes that some stacks suppress extra acknowledgments of the SYN when SYN-ACK carries data from Richard Scheffenegger, and adds MAY-18 numbering based on note from Jonathan Morton.

The -22 revision includes small clarifications on terminology (might versus may) and IPv6 extension headers versus IPv4 options, based on comments from Gorry Fairhurst.

The -23 revision has a fix to indentation from Michael Tuexen and idnits issues addressed from Michael Scharf.

The -24 revision incorporates changes after Martin Duke's AD review, including further feedback on those comments from Yuchung Cheng and Joe Touch. Important changes for review include (1) removal of the need to check for the PUSH flag when evaluating the SWS override timer expiration, (2) clarification about receding urgent pointer, and (3) de-duplicating handling of the RST checking between step 4 and step 1.

The -25 revision incorporates changes based on the GENART review from Francis Dupont, SECDIR review from Kyle Rose, and OPSDIR review from Sarah Banks.

The -26 revision incorporates changes stemming from the IESG reviews, and INTDIR review from Bernie Volz.

The -27 revision fixes a few small editorial incompatibilities that Stephen McQuistin found related to automated code generation.

The -28 revision addresses some COMMENTS from Ben Kaduk's IESG review.

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

1. Tony Sabatini's suggestion for describing DO field
2. Per discussion with Joe Touch (TAPS list, 6/20/2015), the description of the API could be revisited
3. Reducing the R2 value for SYNs has been suggested as a possible topic for future consideration.

Early in the process of updating RFC 793, Scott Brim mentioned that this should include a PERPASS/privacy review. This may be something for the chairs or AD to request during WGLC or IETF LC.

6. IANA Considerations

In the "Transmission Control Protocol (TCP) Header Flags" registry, IANA is asked to make several changes described in this section.

RFC 3168 originally created this registry, but only populated it with the new bits defined in RFC 3168, neglecting the other bits that had previously been described in RFC 793 and other documents. Bit 7 has since also been updated by RFC 8311.

The "Bit" column is renamed below as the "Bit Offset" column, since it references each header flag's offset within the 16-bit aligned view of the TCP header in Figure 1. The bits in offsets 0 through 4 are the TCP segment Data Offset field, and not header flags.

IANA should add a column for "Assignment Notes".

IANA should assign values indicated below.

TCP Header Flags

Bit Notes Offset	Name	Reference	Assignment
---	----	-----	-----
4	Reserved for future use	(this document)	
5	Reserved for future use	(this document)	
6	Reserved for future use	(this document)	
7	Reserved for future use	[RFC8311]	[1]
8	CWR (Congestion Window Reduced)	[RFC3168]	
9	ECE (ECN-Echo)	[RFC3168]	
10	Urgent Pointer field is significant (URG)	(this document)	
11	Acknowledgment field is significant (ACK)	(this document)	
12	Push Function (PSH)	(this document)	
13	Reset the connection (RST)	(this document)	
14	Synchronize sequence numbers (SYN)	(this document)	
15	No more data from sender (FIN)	(this document)	

FOOTNOTES:

[1] Previously used by Historic [RFC3540] as NS (Nonce Sum).

This TCP Header Flags registry should also be moved to a sub-registry under the global "Transmission Control Protocol (TCP) Parameters registry (<https://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml>).

The registry's Registration Procedure should remain Standards Action, but the Reference can be updated to this document, and the Note removed.

7. Security and Privacy Considerations

The TCP design includes only rudimentary security features that improve the robustness and reliability of connections and application data transfer, but there are no built-in cryptographic capabilities to support any form of confidentiality, authentication, or other typical security functions. Non-cryptographic enhancements (e.g. [9]) have been developed to improve robustness of TCP connections to particular types of attacks, but the applicability and protections of non-cryptographic enhancements are limited (e.g. see section 1.1 of [9]). Applications typically utilize lower-layer (e.g. IPsec) and upper-layer (e.g. TLS) protocols to provide security and privacy for TCP connections and application data carried in TCP. Methods based on TCP options have been developed as well, to support some security capabilities.

In order to fully provide confidentiality, integrity protection, and authentication for TCP connections (including their control flags) IPsec is the only current effective method. For integrity protection and authentication, the TCP Authentication Option (TCP-AO) [39] is available, with a proposed extension to also provide confidentiality for the segment payload. Other methods discussed in this section may provide confidentiality or integrity protection for the payload, but for the TCP header only cover either a subset of the fields (e.g. tcpcrypt [57]) or none at all (e.g. TLS). Other security features that have been added to TCP (e.g. ISN generation, sequence number checks, and others) are only capable of partially hindering attacks.

Applications using long-lived TCP flows have been vulnerable to attacks that exploit the processing of control flags described in earlier TCP specifications [34]. TCP-MD5 was a commonly implemented TCP option to support authentication for some of these connections, but had flaws and is now deprecated. TCP-AO provides a capability to protect long-lived TCP connections from attacks, and has superior properties to TCP-MD5. It does not provide any privacy for application data, nor for the TCP headers.

The "tcpcrypt" [57] Experimental extension to TCP provides the ability to cryptographically protect connection data. Metadata aspects of the TCP flow are still visible, but the application stream is well-protected. Within the TCP header, only the urgent pointer and FIN flag are protected through tcpcrypt.

The TCP Roadmap [50] includes notes about several RFCs related to TCP security. Many of the enhancements provided by these RFCs have been integrated into the present document, including ISN generation, mitigating blind in-window attacks, and improving handling of soft errors and ICMP packets. These are all discussed in greater detail in the referenced RFCs that originally described the changes needed to earlier TCP specifications. Additionally, see RFC 6093 [40] for discussion of security considerations related to the urgent pointer field, that has been deprecated.

Since TCP is often used for bulk transfer flows, some attacks are possible that abuse the TCP congestion control logic. An example is "ACK-division" attacks. Updates that have been made to the TCP congestion control specifications include mechanisms like Appropriate Byte Counting (ABC) [30] that act as mitigations to these attacks.

Other attacks are focused on exhausting the resources of a TCP server. Examples include SYN flooding [33] or wasting resources on non-progressing connections [42]. Operating systems commonly implement mitigations for these attacks. Some common defenses also utilize proxies, stateful firewalls, and other technologies outside the end-host TCP implementation.

The concept of a protocol's "wire image" is described in RFC 8546 [56], which describes how TCP's cleartext headers expose more metadata to nodes on the path than is strictly required to route the packets to their destination. On-path adversaries may be able to leverage this metadata. Lessons learned in this respect from TCP have been applied in the design of newer transports like QUIC [60]. Additionally, based partly on experiences with TCP and its extensions, there are considerations that might be applicable for future TCP extensions and other transports that the IETF has documented in RFC 9065 [61], along with IAB recommendations in RFC 8558 [58] and [68].

There are also methods of "fingerprinting" that can be used to infer the host TCP implementation (operating system) version or platform information. These collect observations of several aspects such as the options present in segments, the ordering of options, the specific behaviors in the case of various conditions, packet timing, packet sizing, and other aspects of the protocol that are left to be determined by an implementer, and can use those observations to identify information about the host and implementation.

8. Acknowledgements

This document is largely a revision of RFC 793, which Jon Postel was the editor of. Due to his excellent work, it was able to last for three decades before we felt the need to revise it.

Andre Oppermann was a contributor and helped to edit the first revision of this document.

We are thankful for the assistance of the IETF TCPM working group chairs, over the course of work on this document:

Michael Scharf

Yoshifumi Nishida

Pasi Sarolahti

Michael Tuexen

During the discussions of this work on the TCPM mailing list, in working group meetings, and via area reviews, helpful comments, critiques, and reviews were received from (listed alphabetically by last name): Praveen Balasubramanian, David Borman, Mohamed Boucadair, Bob Briscoe, Neal Cardwell, Yuchung Cheng, Martin Duke, Francis Dupont, Ted Faber, Gorrry Fairhurst, Fernando Gont, Rodney Grimes, Yi Huang, Rahul Jadhav, Markku Kojo, Mike Kosek, Juhamatti Kuusisaari, Kevin Lahey, Kevin Mason, Matt Mathis, Stephen McQuistin, Jonathan Morton, Matt Olson, Tommy Pauly, Tom Petch, Hagen Paul Pfeifer, Kyle Rose, Anthony Sabatini, Michael Scharf, Greg Skinner, Joe Touch, Michael Tuexen, Reji Varghese, Bernie Volz, Tim Wicinski, Lloyd Wood, and Alex Zimmermann.

Joe Touch provided additional help in clarifying the description of segment size parameters and PMTUD/PLPMTUD recommendations. Markku Kojo helped put together the text in the section on TCP Congestion Control.

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, Charles Deng, Merlin Buge.

9. References

9.1. Normative References

- [1] Postel, J., "Internet Protocol", STD 5, RFC 791, DOI 10.17487/RFC0791, September 1981, <<https://www.rfc-editor.org/info/rfc791>>.
- [2] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, DOI 10.17487/RFC1191, November 1990, <<https://www.rfc-editor.org/info/rfc1191>>.
- [3] 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>>.
- [4] Nichols, K., Blake, S., Baker, F., and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", RFC 2474, DOI 10.17487/RFC2474, December 1998, <<https://www.rfc-editor.org/info/rfc2474>>.

- [5] Floyd, S., "Congestion Control Principles", BCP 41, RFC 2914, DOI 10.17487/RFC2914, September 2000, <<https://www.rfc-editor.org/info/rfc2914>>.
- [6] 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>>.
- [7] Floyd, S. and M. Allman, "Specifying New Congestion Control Algorithms", BCP 133, RFC 5033, DOI 10.17487/RFC5033, August 2007, <<https://www.rfc-editor.org/info/rfc5033>>.
- [8] 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>>.
- [9] Ramaiah, A., Stewart, R., and M. Dalal, "Improving TCP's Robustness to Blind In-Window Attacks", RFC 5961, DOI 10.17487/RFC5961, August 2010, <<https://www.rfc-editor.org/info/rfc5961>>.
- [10] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, DOI 10.17487/RFC6298, June 2011, <<https://www.rfc-editor.org/info/rfc6298>>.
- [11] Gont, F., "Deprecation of ICMP Source Quench Messages", RFC 6633, DOI 10.17487/RFC6633, May 2012, <<https://www.rfc-editor.org/info/rfc6633>>.
- [12] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [13] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", STD 86, RFC 8200, DOI 10.17487/RFC8200, July 2017, <<https://www.rfc-editor.org/info/rfc8200>>.
- [14] McCann, J., Deering, S., Mogul, J., and R. Hinden, Ed., "Path MTU Discovery for IP version 6", STD 87, RFC 8201, DOI 10.17487/RFC8201, July 2017, <<https://www.rfc-editor.org/info/rfc8201>>.

- [15] Allman, M., "Requirements for Time-Based Loss Detection", BCP 233, RFC 8961, DOI 10.17487/RFC8961, November 2020, <<https://www.rfc-editor.org/info/rfc8961>>.

9.2. Informative References

- [16] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <<https://www.rfc-editor.org/info/rfc793>>.
- [17] Postel, J., "The TCP Maximum Segment Size and Related Topics", RFC 879, DOI 10.17487/RFC0879, November 1983, <<https://www.rfc-editor.org/info/rfc879>>.
- [18] Nagle, J., "Congestion Control in IP/TCP Internetworks", RFC 896, DOI 10.17487/RFC0896, January 1984, <<https://www.rfc-editor.org/info/rfc896>>.
- [19] Reynolds, J. and J. Postel, "Official Internet protocols", RFC 1011, DOI 10.17487/RFC1011, May 1987, <<https://www.rfc-editor.org/info/rfc1011>>.
- [20] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <<https://www.rfc-editor.org/info/rfc1122>>.
- [21] Almquist, P., "Type of Service in the Internet Protocol Suite", RFC 1349, DOI 10.17487/RFC1349, July 1992, <<https://www.rfc-editor.org/info/rfc1349>>.
- [22] Braden, R., "T/TCP -- TCP Extensions for Transactions Functional Specification", RFC 1644, DOI 10.17487/RFC1644, July 1994, <<https://www.rfc-editor.org/info/rfc1644>>.
- [23] Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, "TCP Selective Acknowledgment Options", RFC 2018, DOI 10.17487/RFC2018, October 1996, <<https://www.rfc-editor.org/info/rfc2018>>.
- [24] Paxson, V., Allman, M., Dawson, S., Fenner, W., Griner, J., Heavens, I., Lahey, K., Semke, J., and B. Volz, "Known TCP Implementation Problems", RFC 2525, DOI 10.17487/RFC2525, March 1999, <<https://www.rfc-editor.org/info/rfc2525>>.

- [25] Borman, D., Deering, S., and R. Hinden, "IPv6 Jumbograms", RFC 2675, DOI 10.17487/RFC2675, August 1999, <<https://www.rfc-editor.org/info/rfc2675>>.
- [26] Xiao, X., Hannan, A., Paxson, V., and E. Crabbe, "TCP Processing of the IPv4 Precedence Field", RFC 2873, DOI 10.17487/RFC2873, June 2000, <<https://www.rfc-editor.org/info/rfc2873>>.
- [27] Floyd, S., Mahdavi, J., Mathis, M., and M. Podolsky, "An Extension to the Selective Acknowledgement (SACK) Option for TCP", RFC 2883, DOI 10.17487/RFC2883, July 2000, <<https://www.rfc-editor.org/info/rfc2883>>.
- [28] Lahey, K., "TCP Problems with Path MTU Discovery", RFC 2923, DOI 10.17487/RFC2923, September 2000, <<https://www.rfc-editor.org/info/rfc2923>>.
- [29] Balakrishnan, H., Padmanabhan, V., Fairhurst, G., and M. Sooriyabandara, "TCP Performance Implications of Network Path Asymmetry", BCP 69, RFC 3449, DOI 10.17487/RFC3449, December 2002, <<https://www.rfc-editor.org/info/rfc3449>>.
- [30] Allman, M., "TCP Congestion Control with Appropriate Byte Counting (ABC)", RFC 3465, DOI 10.17487/RFC3465, February 2003, <<https://www.rfc-editor.org/info/rfc3465>>.
- [31] Fenner, B., "Experimental Values In IPv4, IPv6, ICMPv4, ICMPv6, UDP, and TCP Headers", RFC 4727, DOI 10.17487/RFC4727, November 2006, <<https://www.rfc-editor.org/info/rfc4727>>.
- [32] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, DOI 10.17487/RFC4821, March 2007, <<https://www.rfc-editor.org/info/rfc4821>>.
- [33] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", RFC 4987, DOI 10.17487/RFC4987, August 2007, <<https://www.rfc-editor.org/info/rfc4987>>.
- [34] Touch, J., "Defending TCP Against Spoofing Attacks", RFC 4953, DOI 10.17487/RFC4953, July 2007, <<https://www.rfc-editor.org/info/rfc4953>>.
- [35] Culley, P., Elzur, U., Recio, R., Bailey, S., and J. Carrier, "Marker PDU Aligned Framing for TCP Specification", RFC 5044, DOI 10.17487/RFC5044, October 2007, <<https://www.rfc-editor.org/info/rfc5044>>.

- [36] Gont, F., "TCP's Reaction to Soft Errors", RFC 5461, DOI 10.17487/RFC5461, February 2009, <<https://www.rfc-editor.org/info/rfc5461>>.
- [37] StJohns, M., Atkinson, R., and G. Thomas, "Common Architecture Label IPv6 Security Option (CALIPSO)", RFC 5570, DOI 10.17487/RFC5570, July 2009, <<https://www.rfc-editor.org/info/rfc5570>>.
- [38] Sandlund, K., Pelletier, G., and L-E. Jonsson, "The RObust Header Compression (ROHC) Framework", RFC 5795, DOI 10.17487/RFC5795, March 2010, <<https://www.rfc-editor.org/info/rfc5795>>.
- [39] Touch, J., Mankin, A., and R. Bonica, "The TCP Authentication Option", RFC 5925, DOI 10.17487/RFC5925, June 2010, <<https://www.rfc-editor.org/info/rfc5925>>.
- [40] Gont, F. and A. Yourtchenko, "On the Implementation of the TCP Urgent Mechanism", RFC 6093, DOI 10.17487/RFC6093, January 2011, <<https://www.rfc-editor.org/info/rfc6093>>.
- [41] Gont, F., "Reducing the TIME-WAIT State Using TCP Timestamps", BCP 159, RFC 6191, DOI 10.17487/RFC6191, April 2011, <<https://www.rfc-editor.org/info/rfc6191>>.
- [42] Bashyam, M., Jethanandani, M., and A. Ramaiah, "TCP Sender Clarification for Persist Condition", RFC 6429, DOI 10.17487/RFC6429, December 2011, <<https://www.rfc-editor.org/info/rfc6429>>.
- [43] Gont, F. and S. Bellovin, "Defending against Sequence Number Attacks", RFC 6528, DOI 10.17487/RFC6528, February 2012, <<https://www.rfc-editor.org/info/rfc6528>>.
- [44] Borman, D., "TCP Options and Maximum Segment Size (MSS)", RFC 6691, DOI 10.17487/RFC6691, July 2012, <<https://www.rfc-editor.org/info/rfc6691>>.
- [45] Touch, J., "Updated Specification of the IPv4 ID Field", RFC 6864, DOI 10.17487/RFC6864, February 2013, <<https://www.rfc-editor.org/info/rfc6864>>.
- [46] Touch, J., "Shared Use of Experimental TCP Options", RFC 6994, DOI 10.17487/RFC6994, August 2013, <<https://www.rfc-editor.org/info/rfc6994>>.

- [47] McPherson, D., Oran, D., Thaler, D., and E. Osterweil, "Architectural Considerations of IP Anycast", RFC 7094, DOI 10.17487/RFC7094, January 2014, <<https://www.rfc-editor.org/info/rfc7094>>.
- [48] Borman, D., Braden, B., Jacobson, V., and R. Scheffenegger, Ed., "TCP Extensions for High Performance", RFC 7323, DOI 10.17487/RFC7323, September 2014, <<https://www.rfc-editor.org/info/rfc7323>>.
- [49] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<https://www.rfc-editor.org/info/rfc7413>>.
- [50] Duke, M., Braden, R., Eddy, W., Blanton, E., and A. Zimmermann, "A Roadmap for Transmission Control Protocol (TCP) Specification Documents", RFC 7414, DOI 10.17487/RFC7414, February 2015, <<https://www.rfc-editor.org/info/rfc7414>>.
- [51] Black, D., Ed. and P. Jones, "Differentiated Services (Diffserv) and Real-Time Communication", RFC 7657, DOI 10.17487/RFC7657, November 2015, <<https://www.rfc-editor.org/info/rfc7657>>.
- [52] 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>>.
- [53] Fairhurst, G., Ed., Trammell, B., Ed., and M. Kuehlewind, Ed., "Services Provided by IETF Transport Protocols and Congestion Control Mechanisms", RFC 8095, DOI 10.17487/RFC8095, March 2017, <<https://www.rfc-editor.org/info/rfc8095>>.
- [54] Welzl, M., Tuexen, M., and N. Khademi, "On the Usage of Transport Features Provided by IETF Transport Protocols", RFC 8303, DOI 10.17487/RFC8303, February 2018, <<https://www.rfc-editor.org/info/rfc8303>>.
- [55] Chown, T., Loughney, J., and T. Winters, "IPv6 Node Requirements", BCP 220, RFC 8504, DOI 10.17487/RFC8504, January 2019, <<https://www.rfc-editor.org/info/rfc8504>>.
- [56] Trammell, B. and M. Kuehlewind, "The Wire Image of a Network Protocol", RFC 8546, DOI 10.17487/RFC8546, April 2019, <<https://www.rfc-editor.org/info/rfc8546>>.

- [57] Bittau, A., Giffin, D., Handley, M., Mazieres, D., Slack, Q., and E. Smith, "Cryptographic Protection of TCP Streams (tcpcrypt)", RFC 8548, DOI 10.17487/RFC8548, May 2019, <<https://www.rfc-editor.org/info/rfc8548>>.
- [58] Hardie, T., Ed., "Transport Protocol Path Signals", RFC 8558, DOI 10.17487/RFC8558, April 2019, <<https://www.rfc-editor.org/info/rfc8558>>.
- [59] Ford, A., Raiciu, C., Handley, M., Bonaventure, O., and C. Paasch, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 8684, DOI 10.17487/RFC8684, March 2020, <<https://www.rfc-editor.org/info/rfc8684>>.
- [60] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <<https://www.rfc-editor.org/info/rfc9000>>.
- [61] Fairhurst, G. and C. Perkins, "Considerations around Transport Header Confidentiality, Network Operations, and the Evolution of Internet Transport Protocols", RFC 9065, DOI 10.17487/RFC9065, July 2021, <<https://www.rfc-editor.org/info/rfc9065>>.
- [62] IANA, "Transmission Control Protocol (TCP) Parameters, <https://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml>", 2019.
- [63] IANA, "Transmission Control Protocol (TCP) Header Flags, <https://www.iana.org/assignments/tcp-header-flags/tcp-header-flags.xhtml>", 2019.
- [64] Gont, F., "Processing of IP Security/Compartment and Precedence Information by TCP", Work in Progress, Internet-Draft, draft-gont-tcpm-tcp-seccomp-prec-00, 29 March 2012, <<http://www.ietf.org/internet-drafts/draft-gont-tcpm-tcp-seccomp-prec-00.txt>>.
- [65] Gont, F. and D. Borman, "On the Validation of TCP Sequence Numbers", Work in Progress, Internet-Draft, draft-gont-tcpm-tcp-seq-validation-04, 11 March 2019, <<http://www.ietf.org/internet-drafts/draft-gont-tcpm-tcp-seq-validation-04.txt>>.

- [66] Touch, J. and W. Eddy, "TCP Extended Data Offset Option", Work in Progress, Internet-Draft, draft-ietf-tcpm-tcp-edo-10, 19 July 2018, <<http://www.ietf.org/internet-drafts/draft-ietf-tcpm-tcp-edo-10.txt>>.
- [67] McQuistin, S., Band, V., Jacob, D., and C. Perkins, "Describing Protocol Data Units with Augmented Packet Header Diagrams", Work in Progress, Internet-Draft, draft-mcquistin-augmented-ascii-diagrams-08, 5 May 2021, <<https://www.ietf.org/archive/id/draft-mcquistin-augmented-ascii-diagrams-08.txt>>.
- [68] Thomson, M. and T. Pauly, "Long-term Viability of Protocol Extension Mechanisms", Work in Progress, Internet-Draft, draft-iab-use-it-or-lose-it-02, 23 August 2021, <<https://www.ietf.org/archive/id/draft-iab-use-it-or-lose-it-02.txt>>.
- [69] Minshall, G., "A Proposed Modification to Nagle's Algorithm", Work in Progress, Internet-Draft, draft-minshall-nagle-01, June 1999, <<https://datatracker.ietf.org/doc/html/draft-minshall-nagle-01>>.
- [70] Dalal, Y. and C. Sunshine, "Connection Management in Transport Protocols", Computer Networks Vol. 2, No. 6, pp. 454-473, December 1978.
- [71] Faber, T., Touch, J., and W. Yui, "The TIME-WAIT state in TCP and Its Effect on Busy Servers", Proceedings of IEEE INFOCOM pp. 1573-1583, March 1999.
- [72] Postel, J., "Comments on Action Items from the January Meeting", IEN 177, March 1981, <<https://www.rfc-editor.org/ien/ien177.txt>>.
- [73] "Segmentation Offloads", Linux Networking Documentation , <<https://www.kernel.org/doc/html/latest/networking/segmentation-offloads.html>>.

Appendix A. Other Implementation Notes

This section includes additional notes and references on TCP implementation decisions that are currently not a part of the RFC series or included within the TCP standard. These items can be considered by implementers, but there was not yet a consensus to include them in the standard.

A.1. IP Security Compartment and Precedence

The IPv4 specification [1] includes a precedence value in the (now obsolete) Type of Service field (TOS) field. It was modified in [21], and then obsolete by the definition of Differentiated Services (DiffServ) [4]. Setting and conveying TOS between the network layer, TCP implementation, and applications is obsolete, and replaced by DiffServ in the current TCP specification.

RFC 793 required checking the IP security compartment and precedence on incoming TCP segments for consistency within a connection, and with application requests. Each of these aspects of IP have become outdated, without specific updates to RFC 793. The issues with precedence were fixed by [26], which is Standards Track, and so this present TCP specification includes those changes. However, the state of IP security options that may be used by MLS systems is not as apparent in the IETF currently.

Resetting connections when incoming packets do not meet expected security compartment or precedence expectations has been recognized as a possible attack vector [64], and there has been discussion about amending the TCP specification to prevent connections from being aborted due to non-matching IP security compartment and DiffServ codepoint values.

A.1.1. Precedence

In DiffServ the former precedence values are treated as Class Selector codepoints, and methods for compatible treatment are described in the DiffServ architecture. The RFC 793/1122 TCP specification includes logic intending to have connections use the highest precedence requested by either endpoint application, and to keep the precedence consistent throughout a connection. This logic from the obsolete TOS is not applicable for DiffServ, and should not be included in TCP implementations, though changes to DiffServ values within a connection are discouraged. For discussion of this, see RFC 7657 (sec 5.1, 5.3, and 6) [51].

The obsolete TOS processing rules in TCP assumed bidirectional (or symmetric) precedence values used on a connection, but the DiffServ architecture is asymmetric. Problems with the old TCP logic in this regard were described in [26] and the solution described is to ignore IP precedence in TCP. Since RFC 2873 is a Standards Track document (although not marked as updating RFC 793), current implementations are expected to be robust to these conditions. Note that the DiffServ field value used in each direction is a part of the interface between TCP and the network layer, and values in use can be indicated both ways between TCP and the application.

A.1.2. MLS Systems

The IP security option (IPSO) and compartment defined in [1] was refined in RFC 1038 that was later obsoleted by RFC 1108. The Commercial IP Security Option (CIPSO) is defined in FIPS-188 (withdrawn by NIST in 2015), and is supported by some vendors and operating systems. RFC 1108 is now Historic, though RFC 791 itself has not been updated to remove the IP security option. For IPv6, a similar option (CALIPSO) has been defined [37]. RFC 793 includes logic that includes the IP security/compartment information in treatment of TCP segments. References to the IP "security/compartment" in this document may be relevant for Multi-Level Secure (MLS) system implementers, but can be ignored for non-MLS implementations, consistent with running code on the Internet. See Appendix A.1 for further discussion. Note that RFC 5570 describes some MLS networking scenarios where IPSO, CIPSO, or CALIPSO may be used. In these special cases, TCP implementers should see section 7.3.1 of RFC 5570, and follow the guidance in that document.

A.2. Sequence Number Validation

There are cases where the TCP sequence number validation rules can prevent ACK fields from being processed. This can result in connection issues, as described in [65], which includes descriptions of potential problems in conditions of simultaneous open, self-connects, simultaneous close, and simultaneous window probes. The document also describes potential changes to the TCP specification to mitigate the issue by expanding the acceptable sequence numbers.

In Internet usage of TCP, these conditions are rarely occurring. Common operating systems include different alternative mitigations, and the standard has not been updated yet to codify one of them, but implementers should consider the problems described in [65].

A.3. Nagle Modification

In common operating systems, both the Nagle algorithm and delayed acknowledgements are implemented and enabled by default. TCP is used by many applications that have a request-response style of communication, where the combination of the Nagle algorithm and delayed acknowledgements can result in poor application performance. A modification to the Nagle algorithm is described in [69] that improves the situation for these applications.

This modification is implemented in some common operating systems, and does not impact TCP interoperability. Additionally, many applications simply disable Nagle, since this is generally supported by a socket option. The TCP standard has not been updated to include this Nagle modification, but implementers may find it beneficial to consider.

A.4. Low Watermark Settings

Some operating system kernel TCP implementations include socket options that allow specifying the number of bytes in the buffer until the socket layer will pass sent data to TCP (SO_SNDLOWAT) or to the application on receiving (SO_RCVLOWAT).

In addition, another socket option (TCP_NOTSENT_LOWAT) can be used to control the amount of unsent bytes in the write queue. This can help a sending TCP application to avoid creating large amounts of buffered data (and corresponding latency). As an example, this may be useful for applications that are multiplexing data from multiple upper level streams onto a connection, especially when streams may be a mix of interactive / real-time and bulk data transfer.

Appendix B. TCP Requirement Summary

This section is adapted from RFC 1122.

Note that there is no requirement related to PLPMTUD in this list, but that PLPMTUD is recommended.

FEATURE	ReqID	MUST	SHOULD	RECOMMENDED	OPTIONAL	NOT RECOMMENDED	Forbidden
Push flag							
Aggregate or queue un-pushed data	MAY-16			x			
Sender collapse successive PSH flags	SHLD-27		x				
SEND call can specify PUSH	MAY-15			x			
If cannot: sender buffer indefinitely	MUST-60						x
If cannot: PSH last segment	MUST-61	x					
Notify receiving ALP of PSH	MAY-17			x			1

Send max size segment when possible	SHLD-28	x				
Window						
Treat as unsigned number	MUST-1	x				
Handle as 32-bit number	REC-1		x			
Shrink window from right	SHLD-14			x		
- Send new data when window shrinks	SHLD-15			x		
- Retransmit old unacked data within window	SHLD-16	x				
- Time out conn for data past right edge	SHLD-17			x		
Robust against shrinking window	MUST-34	x				
Receiver's window closed indefinitely	MAY-8		x			
Use standard probing logic	MUST-35	x				
Sender probe zero window	MUST-36	x				
First probe after RTO	SHLD-29		x			
Exponential backoff	SHLD-30		x			
Allow window stay zero indefinitely	MUST-37	x				
Retransmit old data beyond SND.UNA+SND.WND	MAY-7		x			
Process RST and URG even with zero window	MUST-66	x				
Urgent Data						
Include support for urgent pointer	MUST-30	x				
Pointer indicates first non-urgent octet	MUST-62	x				
Arbitrary length urgent data sequence	MUST-31	x				
Inform ALP asynchronously of urgent data	MUST-32	x				1
ALP can learn if/how much urgent data Q'd	MUST-33	x				1
ALP employ the urgent mechanism	SHLD-13			x		
TCP Options						
Support the mandatory option set	MUST-4	x				
Receive TCP option in any segment	MUST-5	x				
Ignore unsupported options	MUST-6	x				
Include length for all options except EOL+NOP	MUST-68	x				
Cope with illegal option length	MUST-7	x				
Process options regardless of word alignment	MUST-64	x				
Implement sending & receiving MSS option	MUST-14	x				
IPv4 Send MSS option unless 536	SHLD-5		x			
IPv6 Send MSS option unless 1220	SHLD-5		x			
Send MSS option always	MAY-3			x		
IPv4 Send-MSS default is 536	MUST-15	x				
IPv6 Send-MSS default is 1220	MUST-15	x				
Calculate effective send seg size	MUST-16	x				
MSS accounts for varying MTU	SHLD-6		x			
MSS not sent on non-SYN segments	MUST-65				x	
MSS value based on MMS_R	MUST-67	x				
Pad with zero	MUST-69	x				
TCP Checksums						
Sender compute checksum	MUST-2	x				

Receiver check checksum	MUST-3	x					
ISN Selection							
Include a clock-driven ISN generator component	MUST-8	x					
Secure ISN generator with a PRF component	SHLD-1		x				
PRF computable from outside the host	MUST-9					x	
Opening Connections							
Support simultaneous open attempts	MUST-10	x					
SYN-RECEIVED remembers last state	MUST-11	x					
Passive Open call interfere with others	MUST-41					x	
Function: simultan. LISTENS for same port	MUST-42	x					
Ask IP for src address for SYN if necc.	MUST-44	x					
Otherwise, use local addr of conn.	MUST-45	x					
OPEN to broadcast/multicast IP Address	MUST-46					x	
Silently discard seg to bcast/mcast addr	MUST-57	x					
Closing Connections							
RST can contain data	SHLD-2		x				
Inform application of aborted conn	MUST-12	x					
Half-duplex close connections	MAY-1			x			
Send RST to indicate data lost	SHLD-3		x				
In TIME-WAIT state for 2MSL seconds	MUST-13	x					
Accept SYN from TIME-WAIT state	MAY-2			x			
Use Timestamps to reduce TIME-WAIT	SHLD-4		x				
Retransmissions							
Implement exponential backoff, slow start, and congestion avoidance	MUST-19	x					
Retransmit with same IP ident	MAY-4			x			
Karn's algorithm	MUST-18	x					
Generating ACKs:							
Aggregate whenever possible	MUST-58	x					
Queue out-of-order segments	SHLD-31		x				
Process all Q'd before send ACK	MUST-59	x					
Send ACK for out-of-order segment	MAY-13			x			
Delayed ACKs	SHLD-18		x				
Delay < 0.5 seconds	MUST-40	x					
Every 2nd full-sized segment or 2*RMSS ACK'd	SHLD-19		x				
Receiver SWS-Avoidance Algorithm	MUST-39	x					
Sending data							
Configurable TTL	MUST-49	x					
Sender SWS-Avoidance Algorithm	MUST-38	x					
Nagle algorithm	SHLD-7		x				
Application can disable Nagle algorithm	MUST-17	x					

Connection Failures:

Negative advice to IP on R1 retxs
 Close connection on R2 retxs
 ALP can set R2
 Inform ALP of $R1 \leq \text{retxs} < R2$
 Recommended value for R1
 Recommended value for R2
 Same mechanism for SYNs
 R2 at least 3 minutes for SYN

MUST-20	x					
MUST-20	x					
MUST-21	x					1
SHLD-9		x				1
SHLD-10		x				
SHLD-11		x				
MUST-22	x					
MUST-23	x					

Send Keep-alive Packets:

- Application can request
- Default is "off"
- Only send if idle for interval
- Interval configurable
- Default at least 2 hrs.
- Tolerant of lost ACKs
- Send with no data
- Configurable to send garbage octet

MAY-5			x			
MUST-24	x					
MUST-25	x					
MUST-26	x					
MUST-27	x					
MUST-28	x					
MUST-29	x					
SHLD-12		x				
MAY-6			x			

IP Options

Ignore options TCP doesn't understand
 Time Stamp support
 Record Route support
 Source Route:
 ALP can specify
 Overrides src rt in datagram
 Build return route from src rt
 Later src route overrides

MUST-50	x					
MAY-10			x			
MAY-11			x			
MUST-51	x					1
MUST-52	x					
MUST-53	x					
SHLD-24		x				

Receiving ICMP Messages from IP

Dest. Unreach (0,1,5) => inform ALP
 Abort on Dest. Unreach (0,1,5) =>nn
 Dest. Unreach (2-4) => abort conn
 Source Quench => silent discard
 Abort on Time Exceeded =>
 Abort on Param Problem =>

MUST-54	x					
SHLD-25		x				
MUST-56					x	
SHLD-26		x				
MUST-55	x					
MUST-56					x	
MUST-56					x	

Address Validation

Reject OPEN call to invalid IP address
 Reject SYN from invalid IP address
 Silently discard SYN to bcast/mcast addr

MUST-46	x					
MUST-63	x					
MUST-57	x					

TCP/ALP Interface Services

Error Report mechanism
 ALP can disable Error Report Routine
 ALP can specify DiffServ field for sending
 Passed unchanged to IP

MUST-47	x					
SHLD-20		x				
MUST-48	x					
SHLD-22		x				

ALP can change DiffServ field during connection	SHLD-21		x				
ALP generally changing DiffServ during conn.	SHLD-23				x		
Pass received DiffServ field up to ALP	MAY-9			x			
FLUSH call	MAY-14			x			
Optional local IP addr parm. in OPEN	MUST-43	x					
RFC 5961 Support:							
Implement data injection protection	MAY-12			x			
Explicit Congestion Notification:							
Support ECN	SHLD-8		x				
Alternative Congestion Control:							
Implement alternative conformant algorithm(s)	MAY-18			x			
-----	-----	-	-	-	-	-	-

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

Author's Address

Wesley M. Eddy (editor)
MTI Systems
United States of America
Email: wes@mti-systems.com

Internet Engineering Task Force
INTERNET-DRAFT
File: draft-ietf-tcpm-rto-consider-17.txt
Intended Status: Best Current Practice
Expires: January 27, 2021

M. Allman
ICSI
July 27, 2020

Requirements for Time-Based Loss Detection

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/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 January 27, 2021.

Copyright Notice

Copyright (c) 2020 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

Many protocols must detect packet loss for various reasons (e.g., to ensure reliability using retransmissions or to understand the level of congestion along a network path). While many mechanisms have been designed to detect loss, ultimately, protocols can only count on the passage of time without delivery confirmation to declare a packet "lost". Each implementation of a time-based loss detection mechanism represents a balance between correctness and timeliness and therefore no implementation suits all situations. This document

provides high-level requirements for time-based loss detectors appropriate for general use in unicast communication across 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", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1 Introduction

As a network of networks, the Internet consists of a large variety of links and systems that support a wide variety of tasks and workloads. The service provided by the network varies from best-effort delivery among loosely connected components to highly predictable delivery within controlled environments (e.g., between physically connected nodes, within a tightly controlled data center). Each path through the network has a set of path properties---e.g., available capacity, delay, packet loss. Given the range of networks that make up the Internet, these properties range from largely static to highly dynamic.

This document provides guidelines for developing an understanding of one path property: packet loss. In particular, we offer guidelines for developing and implementing time-based loss detectors that have been gradually learned over the last several decades. We focus on the general case where the loss properties of a path are (a) unknown a priori and (b) dynamically vary over time. Further, while there are numerous root causes of packet loss, we leverage the conservative notion that loss is an implicit indication of congestion [RFC5681]. While this stance is not always correct, as a general assumption it has historically served us well [Jac88]. As we discuss further in section 2, the guidelines in this document should be viewed as a general default for unicast communication across best-effort networks and not as optimal---or even applicable---for all situations.

Given that packet loss is routine in best-effort networks, loss detection is a crucial activity for many protocols and applications and is generally undertaken for two major reasons:

(1) Ensuring reliable data delivery.

This requires a data sender to develop an understanding of which transmitted packets have not arrived at the receiver. This knowledge allows the sender to retransmit missing data.

(2) Congestion control.

As we mention above, packet loss is often taken as an

implicit indication that the sender is transmitting too fast and is overwhelming some portion of the network path. Data senders can therefore use loss to trigger transmission rate reductions.

Various mechanisms are used to detect losses in a packet stream. Often we use continuous or periodic acknowledgments from the recipient to inform the sender's notion of which pieces of data are missing. However, despite our best intentions and most robust mechanisms we cannot place ultimate faith in receiving such acknowledgments, but can only truly depend on the passage of time. Therefore, our ultimate backstop to ensuring that we detect all loss is a timeout. 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 concludes the data was lost in transit.

The specifics of time-based loss detection schemes represent a tradeoff between correctness and responsiveness. In other words we wish to simultaneously:

- wait long enough to ensure the detection of loss is correct, and
- minimize the amount of delay we impose on applications (before repairing loss) and the network (before we reduce the congestion).

Serving both of these goals is difficult as they pull in opposite directions [AP99]. By not waiting long enough to accurately determine a packet has been lost we may provide a needed retransmission in a timely manner, but risk sending unnecessary ("spurious") retransmissions and needlessly lowering the transmission rate. By waiting long enough that we are unambiguously certain a packet has been lost we cannot repair losses in a timely manner and we risk prolonging network congestion.

Many protocols and applications---such as TCP [RFC6298], SCTP [RFC4960], SIP [RFC3261]---use their own time-based loss detection mechanisms. At this point, our experience leads to a recognition that often specific tweaks that deviate from standardized time-based loss detectors do not materially impact network safety with respect to congestion control [AP99]. Therefore, in this document we outline a set of high-level protocol-agnostic requirements for time-based loss detection. The intent is to provide a safe foundation on which implementations have the flexibility to instantiate mechanisms that best realize their specific goals.

2 Context

This document is different from the way we ideally like to engineer systems. Usually, we strive to understand high-level requirements as a starting point. We then methodically engineer specific protocols, algorithms and systems that meet these requirements. Within the IETF standards process we have derived many time-based

loss detection schemes without benefit from some over-arching requirements document---because we had no idea how to write such a document! Therefore, we made the best specific decisions we could in response to specific needs.

At this point, however, the community's experience has matured to the point where we can define a set of general, high-level requirements for time-based loss detection schemes. We now understand how to separate the strategies these mechanisms use that are crucial for network safety from those small details that do not materially impact network safety. The requirements in this document may not be appropriate in all cases. In particular, the guidelines in section 4 are concerned with the general case, but specific situations may allow for more flexibility in terms of loss detection because specific facets of the environment are known (e.g., when operating over a single physical link or within a tightly controlled data center). Therefore, variants, deviations or wholly different time-based loss detectors may be necessary or useful in some cases. The correct way to view this document is as the default case and not as a one-size-fits-all that is optimal in all cases.

Adding a requirements umbrella to a body of existing specifications is inherently messy and we run the risk of creating inconsistencies with both past and future mechanisms. Therefore, we make the following statements about the relationship of this document to past and future specifications:

- This document does not update or obsolete any existing RFC. These previous specifications---while generally consistent with the requirements in this document---reflect community consensus and this document does not change that consensus.
- The requirements in this document are meant to provide for network safety and, as such, SHOULD be used by all future time-based loss detection mechanisms.
- The requirements in this document may not be appropriate in all cases and, therefore, deviations and variants may be necessary in the future (hence the "SHOULD" in the last bullet). However, inconsistencies MUST be (a) explained and (b) gather consensus.

3 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 4:

- (S.1) While there are a bevy of uses for timers in protocols---from rate-based pacing to connection failure detection and beyond---this document is focused only on loss detection.
- (S.2) The requirements for time-based loss detection mechanisms in this document are for the primary or "last resort" loss detection mechanism whether the mechanism is the sole loss

repair strategy or works in concert with other mechanisms.

While a straightforward time-based loss detector is sufficient for simple protocols like DNS [RFC1034,RFC1035], more complex protocols often use more advanced loss detectors to aid performance. For instance, TCP and SCTP have methods to detect (and repair) loss based on explicit endpoint state sharing [RFC2018,RFC4960,RFC6675]. Such mechanisms often provide more timely and precise loss detection than time-based loss detectors. However, these mechanisms do not obviate the need for a "retransmission timeout" or "RTO" because---as we discuss in Section 1---only the passage of time can ultimately be relied upon to detect loss. In other words, ultimately we cannot count on acknowledgments to arrive at the data sender to indicate which packets never arrived at the receiver. In cases such as these we need a time-based loss detector to functions as a "last resort".

Also, note, that some recent proposals have incorporated time as a component of advanced loss detection methods---either as an aggressive first loss detector in certain situations or in conjunction with endpoint state sharing [DCCM13,CCDJ20,IS20]. While these mechanisms can aid timely loss recovery, the protocol ultimately leans on another more conservative timer to ensure reliability when these mechanisms break down. The requirements in this document are only directly applicable to last resort loss detection. However, we expect that many of the requirements can serve as useful guidelines for more aggressive non-last resort timers, as well.

- (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 addresses B and C, A needs to use independent time-based loss detector instances for traffic sent to B and C.

- (S.4) There are cases where state is shared across connections or flows (e.g., [RFC2140], [RFC3124]). State pertaining to time-based loss detection is often discussed as sharable. These situations raise issues that the simple flow-oriented time-based loss detection 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 4 are likely applicable, sharing time-based loss detection information across flows is outside the scope of this document.

4 Requirements

We now list the requirements that apply when designing primary or last resort time-based loss detection mechanisms. For historical reasons and ease of exposition, we refer to the time between sending a packet and determining the packet has been lost due to lack of delivery confirmation as the "retransmission timeout" or "RTO". After the RTO passes without delivery confirmation, the sender may safely assume the packet is lost. However, as discussed above, the detected loss need not be repaired (i.e., the loss could be detected only for congestion control and not reliability purposes).

- (1) As we note above, loss detection happens when a sender does not receive delivery confirmation within some expected period of time. In the absence of any knowledge about the latency of a path, the initial RTO MUST be conservatively set to no less than 1 second.

Correctness is of the utmost importance when transmitting into a network with unknown properties because:

- Premature loss detection can trigger spurious retransmits that could cause issues when a network is already congested.
- Premature loss detection can needlessly cause congestion control to dramatically lower the sender's allowed transmission rate---especially since the rate is already likely low at this stage of the communication. Recovering from such a rate change can take a relatively long time.
- Finally, as discussed below, sometimes using time-based loss detection and retransmissions can cause 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) We now specify four requirements that pertain to setting an expected time interval for delivery confirmation.

Often measuring the time required for delivery confirmation is framed as assessing the "round-trip time (RTT)" of the network path. The RTT 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 misleading and the expected latency is better framed as the "feedback time" (FT). In other words, the expectation is not always simply a network property, but can include additional time before a sender should reasonably expect a response.

For instance, consider a UDP-based DNS request from a client to a recursive resolver [RFC1035]. 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 network RTT between the client and resolver.

Therefore, we express the requirements in terms of FT. Again, for ease of exposition we use "RTO" to indicate the interval between a packet transmission and the decision the packet has been lost---regardless of whether the packet will be retransmitted.

- (a) The RTO SHOULD be set based on multiple observations of the FT when available.

In other words, the RTO should represent an empirically-derived reasonable amount of time that the sender should wait for delivery confirmation before deciding the given data is lost. Network paths are inherently dynamic and therefore it is crucial to incorporate multiple recent FT samples in the RTO to take into account the delay variation across time.

For example, TCP's RTO [RFC6298] would satisfy this requirement due to its use of an exponentially-weighted moving average (EWMA) to combine multiple FT samples into a "smoothed RTT". In the name of conservativeness, TCP goes further to also include an explicit variance term when computing the RTO.

While multiple FT samples are crucial for capturing the delay dynamics of a path, we explicitly do not tightly specify the process---including the number of FT samples to use and how/when to age samples out of the RTO calculation---as the particulars could depend on the situation and/or goals of each specific loss detector.

Finally, FT samples come from packet exchanges between peers. We encourage protocol designers---especially for new protocols---to strive to ensure the feedback is not easily spoofable by on- or off-path attackers such that they can perturb a host's notion of the FT. Ideally, all messages would be cryptographically secure, but given that this is not always possible---especially in legacy protocols---using a healthy amount of randomness in the packets is encouraged.

- (b) FT observations SHOULD be taken and incorporated into the RTO at least once per RTT or as frequently as data is exchanged in cases where that happens less frequently than once per RTT.

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.

As an example, 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 non-data exchanges for various reasons---e.g., keepalives, heartbeats, control messages. To the extent that the latency of these exchanges 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 packet 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 NOT use either version of the FT sample and hence not update the RTO (as discussed in [KP87,RFC6298]).

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 packets 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) Loss detected by the RTO mechanism MUST be taken as an indication of network congestion and the sending rate adapted using a standard mechanism (e.g., TCP collapses the congestion window to one packet [RFC5681]).

This ensures network safety.

An exception to this rule is if an IETF standardized mechanism determines 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, congestion control actions taken based on time-based loss detection could be reversed when a standard mechanism post-facto determines that the cause of the loss was not congestion (e.g., [RFC5682]).

- (4) Each time the RTO is used to detect a loss, 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 either (a) the subsequent successful transmission of non-retransmitted data, or (b) an RTO passes without detecting additional losses. The former will generally be quicker. The latter covers cases where loss is detected, but not repaired.

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

This ensures network safety.

As with guideline (3), an exception to this rule exists if an IETF standardized mechanism determines that a particular loss is not due to congestion.

5 Discussion

We note that research has shown the tension between the responsiveness and correctness of time-based loss detection 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 exponentially weighted moving average (EWMA) gains, lowering the minimum RTO, etc.) can reduce the time required to detect actual loss. However, at the same time, such aggressiveness leads to more cases of mistakenly declaring packets lost that ultimately arrived at the receiver. 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 detecting loss---even if falsely---carries a requirement to invoke a congestion response which will ultimately reduce the transmission rate.

While the tradeoff between responsiveness and correctness seems fundamental, the tradeoff can be made less relevant if the sender can detect and recover from mistaken loss detection. 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 mistakenly declaring packets to be lost.

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 TCP implementations use a steady-state minimum RTO that is 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. This informs the guidelines in the last section (e.g., there is no minimum RTO specified).

Finally, we note that while allowing implementations to be more aggressive could in fact increase the number of needless retransmissions, the above requirements fail safe in that they insist on exponential backoff 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.

6 Security Considerations

This document does not alter the security properties of time-based loss detection mechanisms. See [RFC6298] for a discussion of these within the context of TCP.

7 IANA Considerations

This document has no IANA considerations.

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, Stewart Bryant, Martin Duke, Wesley Eddy, Gorrry Fairhurst, Rahul Arvind Jadhav, Benjamin Kaduk, Mirja Kuhlewind, Nicolas Kuhn, Jonathan Looney and Michael Scharf provided useful comments on previous versions of this document.

Normative References

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

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", RFC 8174, May 2017.

Informative References

[AP99] Allman, M., V. Paxson, "On Estimating End-to-End Network Path Properties", Proceedings of the ACM SIGCOMM Technical Symposium, September 1999.

[CCDJ20] Cheng, Y., N. Cardwell, N. Dukkupati, P. Jha, "RACK: a time-based fast loss detection algorithm for TCP", Internet-Draft draft-ietf-tcpm-rack-08.txt (work in progress), March 2020.

- [DCCM13] Dukkupati, N., N. Cardwell, Y. Cheng, M. Mathis, "Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses", Internet-Draft draft-dukkupati-tcpm-tcp-loss-probe-01.txt (work in progress), February 2013.
- [IS20] Iyengar, I., I. Swett, "QUIC Loss Detection and Congestion Control", Internet-Draft draft-ietf-quic-recovery-27.txt (work in progress), March 2020.
- [Jac88] Jacobson, V., "Congestion Avoidance and Control", ACM SIGCOMM, August 1988.
- [KP87] Karn, P. and C. Partridge, "Improving Round-Trip Time Estimates in Reliable Transport Protocols", SIGCOMM 87.
- [RFC1034] Mockapetris, P. "Domain Names - Concepts and Facilities", RFC 1034, November 1987.
- [RFC1035] Mockapetris, P. "Domain Names - Implementation and Specification", RFC 1035, November 1987.
- [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 3124, 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.
- [RFC4960] Stewart, R., "Stream Control Transmission Protocol", RFC 4960, September 2007.
- [RFC5681] Allman, M., V. Paxson, E. Blanton, "TCP Congestion Control", RFC 5681, September 2009.
- [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",
RFC 5740, November 2009.

[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.

[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: October 2022

J. Touch
Independent Consultant
Wes Eddy
MTI Systems
April 15, 2022

TCP Extended Data Offset Option
draft-ietf-tcpm-tcp-edo-12.txt

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.

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.

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>

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 October 15, 2022.

Copyright Notice

Copyright (c) 2022 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 Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

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][To18]. Some of these other extensions can work in conjunction with EDO (e.g., [To18]).

2. Conventions used in this document

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

In this document, these words will appear with that interpretation only when in ALL CAPS. Lower case uses of these words are not to be interpreted as carrying RFC-2119 significance.

In this document, the characters ">>" preceding an indented line(s) indicates a compliance requirement statement using the key words listed above. This convention aids reviewers in quickly identifying or finding the explicit compliance requirements of this RFC.

3. Motivation

TCP supports headers with a total length of up to 15 32-bit words, as indicated in the 4-bit Data Offset field [RFC793]. This accounts

for a total of 60 bytes, of which the default TCP header fields occupy 20 bytes, leaving 40 bytes for options.

TCP connections already use this option space for a variety of capabilities. These include Maximum Segment Size (MSS) [RFC793], Window Scale (WS) [RFC7323], Timestamp (TS) [RFC7323], Selective Acknowledgement (SACK) [RFC2018][RFC6675], TCP Authentication Option (TCP-AO) [RFC5925], Multipath TCP (MP-TCP) [RFC6824], and TCP User Timeout [RFC5482]. Some options occur only in a SYN or SYN/ACK (MSS, WS), and others vary in size when used in SYN vs. non-SYN segments.

Each of these options consumes space, where some options consuming as much space as available (SACK) and other desired combinations can easily exceed the currently available space. For example, it is not currently possible to use TCP-AO with both TS and MP-TCP in the same non-SYN segment, i.e., to combine accurate round-trip estimation, authentication, and multipath support in the same connection - even though these options can be negotiated during a SYN exchange (10 for TS, 16 for TCP-AO, and 12 for MP-TCP).

TCP EDO is intended to overcome this limitation for non-SYN segments, as well as to increase the space available for SACK blocks. Further discussion of the impact of EDO and existing options is discussed in Section 6.4. Extending SYN segments is much more complicated, as discussed in Section 8.7.

4. Requirements for Extending TCP's Data Offset

The primary goal of extending the TCP Data Offset field is to increase the space available for TCP options in all segments except the initial SYN.

An important requirement of any such extension is that it not impact legacy endpoints. Endpoints seeking to use this new option should not incur additional delay or segment exchanges to connect to either new endpoints supporting this option or legacy endpoints without this option. We call this a "backward downgrade" capability.

An additional consideration of this extension is avoiding user data corruption in the presence of popular network devices, including middleboxes. Consideration of middlebox misbehavior can also interfere with extension in the SYN/ACK.

5. The TCP EDO Option

TCP EDO extends the option space for all segments except the initial SYN (i.e., SYN set and ACK not set) and SYN/ACK response. EDO is

indicated by the TCP option codepoint of EDO-OPT and has two types: EDO Supported and EDO Extension, as discussed in the following subsections.

5.1. EDO Supported

EDO capability is determined in both directions using a single exchange of the EDO Supported option (Figure 1). When EDO is desired on a given connection, the SYN and SYN/ACK segments include the EDO Supported option, which consists of the two required TCP option fields: Kind and Length. The EDO Supported option is used only in the SYN and SYN/ACK segments and only to confirm support for EDO in subsequent segments.

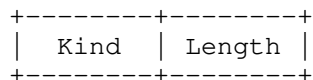


Figure 1 TCP EDO Supported option

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][To18].

The EDO Extension option has negligible impact on other headers because it can either come first or just after security information, and in either case the additional 4 or 6 bytes are easily accommodated within the TCP Data Offset length. Once the EDO option is processed, the entirety of the remainder of the TCP segment is available for any remaining options.

6.5. Connectionless Resets

A RST may arrive during a currently active connection or may be needed to cleanup old state from an abandoned connection. The latter occurs when a new SYN is sent to an endpoint with matching existing connection state, at which point that endpoint responds with a RST and both ends remove stale information.

The EDO Extension option is mandatory on all TCP segments once negotiated, i.e., except in the SYN and SYN/ACK (which establish support) and the RST. A RST may lack the context to know that EDO is active on a connection.

>> The EDO Extension option MAY occur in a RST when the endpoint has connection state that has negotiated EDO. However, unless the RST is generated by an incoming segment that includes an EDO Extension option, the transmitted RST MUST NOT include the EDO Extension option.

6.6. ICMP Handling

ICMP responses are intended to include the IP and the port fields of TCP and UDP headers of typical TCP/IP and UDP/IP packets [RFC792]. This includes the first 8 data bytes of the original datagram, intended to include the transport port numbers used for connection demultiplexing. Later specifications encourage returning as much of the original payload as possible [RFC1812]. In either case, legacy

options or new options in the EDO extension area might or might not be included, and so options are generally not assumed to be part of ICMP processing anyway.

7. Interactions with Middleboxes

Middleboxes are on-path devices that typically examine or modify packets in ways that Internet routers do not [RFC3234]. This includes parsing transport headers and/or rewriting transport segments in ways that may affect EDO.

There are several cases to consider:

- Typical NAT/NAPT devices, which modify only IP address and/or TCP port number fields (with associated TCP checksum updates)
- Middleboxes that try to reconstitute TCP data streams, such as for deep-packet inspection for virus scanning
- Middleboxes that modify known TCP header fields
- Middleboxes that rewrite TCP segments

7.1. Middlebox Coexistence with EDO

Middleboxes can coexist with EDO when they either support EDO or when they ignore its impact on segment structure.

NATs and 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" [Hol1].

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 [Hol1]. 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 NAPT boxes, which modify IP addresses and ports and recompute IPv4 header and TCP segment checksums. Middleboxes that reject unknown options or that process segments in detail without regard for unknown options are not considered; they process segments as if they were an endpoint but do so in ways that are not compliant with existing TCP requirements (e.g., they should have rejected the initial SYN because of its unknown options rather than silently relaying it).

EDO also attempts to avoid creating side-effects, such as might happen if options were split across multiple TCP segments (which could arrive out of order or be lost) or across different TCP connections (which could fail to share fate through firewalls or NAT/NAPTs).

These requirements are similar to those noted in [Ra12], but EDO groups cases of segment modification beyond address and port - such as rewriting, segment drop, sequence number modification, and option stripping - as already in violation of existing TCP requirements regarding unknown options, and so we do not consider their impact on this new option.

8.2. Summary of Approaches

There are three basic ways in which TCP option space extension has been attempted:

1. Use of a TCP option.
2. Redefinition of the existing TCP header fields.
3. Use of option space in multiple TCP segments (split across multiple segments).

A TCP option is the most direct way to extend the option space and is the basis of EDO. This approach cannot extend the option space of the initial SYN.

Redefining existing TCP header fields can be used to either contain additional options or as a pointer indicating alternate ways to interpret the segment payload. All such redefinitions make it difficult to achieve zero-impact backward compatibility, both with legacy endpoints and middleboxes.

Splitting option space across separate segments can create unintended side-effects, such as increased delay to deal with path latency or loss differences.

The following discusses three of the most notable past attempts to extend the TCP option space: Extended Segments, TCPx2, LO/SLO, and LOIC. [Ra12] suggests a few other approaches, including use of TCP option cookies, reuse/overload of other TCP fields (e.g., the URG pointer), or compressing TCP options. None of these is compatible with legacy endpoints or middleboxes.

8.3. Extended Segments

TCP Extended Segments redefined the meaning of currently unused values of the Data Offset (DO) field [Ko04]. TCP defines DO as indicating the length of the TCP header, including options, in 32-bit words. The default TCP header with no options is 5 such words, so the minimum currently valid DO value is 5 (meaning 40 bytes of option space). This document defines interpretations of values 0-4: DO=0 means 48 bytes of option space, DO=1 means 64, DO=2 means 128, DO=3 means 256, and DO=4 means unlimited (e.g., the entire payload is option space). This variant negotiates the use of this capability by using one of these invalid DO values in the initial SYN.

Use of this variant is not backward-compatible with legacy TCP implementations, whether at the desired endpoint or on middleboxes. The variant also defines a way to initiate the feature on the passive side, e.g., using an invalid DO during the SYN/ACK when the initial SYN had a valid DO. This capability allows either side to initiate use of the feature but is also not backward compatible.

8.4. TCPx2

TCPx2 redefines legacy TCP headers by basically doubling all TCP header fields [Al06]. It relies on a new transport protocol number to indicate its use, defeating backward compatibility with all existing TCP capabilities, including firewalls, NATs/NAPTs, and legacy endpoints and applications.

8.5. LO/SLO

The TCP Long Option (LO, [Ed08]) is very similar to EDO, except that presence of LO results in ignoring the existing Data Offset (DO) field and that LO is required to be the first option. EDO considers the need for other fields to be first and declares that the EDO is the last option as indicated by the DO field value. Like LO, EDO is required in every segment once negotiated.

The TCP Long Option draft also specified the SYN Long Option (SLO) [Ed08]. If SLO is used in the initial SYN and successfully negotiated, it is used in each subsequent segment until all of the initial SYN options are transmitted.

LO is backward compatible, as is SLO; in both cases, endpoints not supporting the option would not respond with the option, and in both cases the initial SYN is not itself extended.

SLO does modify the three-way handshake because the connection isn't considered completely established until the first data byte is acknowledged. Legacy TCP can establish a connection even in the absence of data. SLO also changes the semantics of the SYN/ACK; for legacy TCP, this completes the active side connection establishment, where in SLO an additional data ACK is required. A connection whose initial SYN options have been confirmed in the SYN/ACK might still fail upon receipt of additional options sent in later SLO segments. This case - of late negotiation fail - is not addressed in the specification.

8.6. LOIC

TCP Long Options by Invalid Checksum is a dual-stack approach that uses two initial SYNs to initiate all updated connections [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][To18].

Option space cannot be extended in outer layer headers, e.g., IPv4 or IPv6. These layers typically try to avoid extensions altogether, to simplify forwarding processing at routers. Introducing new shim layers to accommodate additional option space would interfere with deep-packet inspection mechanisms that are in widespread use.

As a result, EDO does not attempt to extend the space available for options in TCP initial SYNs. It does extend that space in all other segments (including SYN/ACK), which has always been trivially possible once an option is defined.

9. Implementation Issues

TCP segment processing can involve accessing nonlinear data structures, such as chains of buffers. Such chains are often designed so that the maximum default TCP header (60 bytes) fits in the first buffer. Extending the TCP header across multiple buffers may necessitate buffer traversal functions that span boundaries between buffers. Such traversal can also have a significant performance impact, which is additional rationale for using TCP option space - even extended option space - sparingly.

Although EDO can be large enough to consume the entire segment, it is important to leave space for data so that the TCP connection can make forward progress. It would be wise to limit EDO to consuming no more than MSS-4 bytes of the IP segment, preferably even less (e.g., MSS-128 bytes).

When using the ExID variant for testing and experimentation, either TCP option codepoint (253, 254) is valid in sent or received segments.

Implementers need to be careful about the potential for offload support interfering with this option. The EDO data needs to be passed to the protocol stack as part of the option space, not integrated with the user segment, to allow the offload to independently determine user data segment boundaries and combine them correctly with the extended option data. Some legacy hardware receive offload engines may present challenges in this regard, and

may be incompatible with EDO where they incorrectly attempt to process segments with unknown options. Such offload engines are part of the protocol stack and updated accordingly. Issues with incorrect resegmentation by an offload engine can be detected in the same way as middlebox tampering.

10. Security Considerations

It is meaningless to have the Data Offset further exceed the position of the EDO data offset option.

>> When the EDO Extension option is present, the EDO Extension option SHOULD be the last non-null option covered by the TCP Data Offset, because it would be the last option affected by Data Offset.

This also makes it more difficult to use the Data Offset field as a covert channel.

11. IANA Considerations

We request that, upon publication, this option be assigned a TCP Option codepoint by IANA, which the RFC Editor will replace EDO-OPT in this document with codepoint value.

The TCP Experimental ID (ExID) with a 16-bit value of 0x0ED0 (in network standard byte order) has been assigned for use during testing and preliminary experiments.

12. References

12.1. Normative References

[RFC793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

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

12.2. Informative References

[Al06] Allman, M., "TCPx2: Don't Fence Me In", draft-allman-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.
- [Ho11] Honda, M., Nishida, Y., Raiciu, C., Greenhalgh, A., Handley, M., and H. Tokuda, "Is it still possible to extend TCP", Proc. ACM Sigcomm Internet Measurement Conference (IMC), 2011, pp. 181-194.
- [Ko04] Kohler, E., "Extended Option Space for TCP", draft-kohler-tcpm-extopt-00 (work in progress), September 2004.
- [Ni15] Nishida, Y., "A-PAWS: Alternative Approach for PAWS", draft-nishida-tcpm-apaws-02 (work in progress), Oct. 2015.
- [Ra12] Ramaiah, A., "TCP option space extension", draft-ananth-tcpm-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.
- [To21] Touch, J., T. Faber, "TCP SYN Extended Option Space Using an Out-of-Band Segment", draft-touch-tcpm-tcp-syn-ext-opt (work in progress), Oct. 2019.
- [Yo11] Yourtchenko, A., "Introducing TCP Long Options by Invalid Checksum", draft-yourtchenko-tcp-loic-00 (work in progress), April 2011.

13. Acknowledgments

The authors would like to thank the IETF TCPM WG for their feedback, in particular: Oliver Bonaventure, Bob Briscoe, Ted Faber, John Leslie, Pasi Sarolahti, Richard Scheffenegger, and Alexander Zimmerman.

This work is partly supported by USC/ISI's Postel Center.

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

Authors' Addresses

Joe Touch
Manhattan Beach, CA 90266 USA

Phone: +1 (310) 560-0334
Email: touch@strayalpha.com

Wesley M. Eddy
MTI Systems
US

Email: wes@mti-systems.com

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-segnos-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.

1.3. Backwards Compatibility

This document proposes a mechanism for providing backwards compatibility with existing TCP implementations that only support 32-bit sequence numbers. The document takes advantage of the fact that the least-significant 32-bits of the 64-bit sequence number should have the same properties as the normal 32-bit sequence number: it is the same size, should be seeded to be as random as 32-bit sequence numbers, and should continue to wrap as expected.

1.4. 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 [RFC2119].

2. Extended Sequence Numbers

This document allows the use of 64-bit sequence numbers if both endpoints of a TCP connection agree to use them. If both endpoints agree to use them, the endpoints should store 64 bits of sequence number and acknowledgment number information and should conduct all operations on these values using modulo 2^{64} arithmetic.

Although a host is free to store the 64-bit sequence number information in whatever format it desires, this document make a logical distinction between the most-significant 32 bits and the least-significant 32 bits of sequence number information. This division is represented in Figure 1.

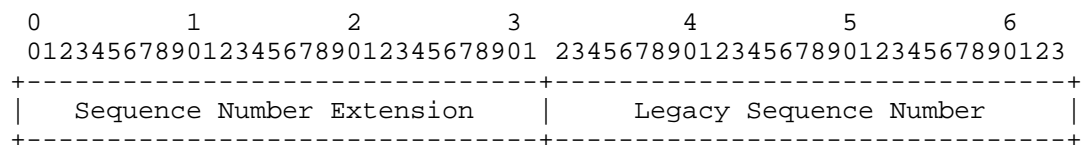


Figure 1: Logical division of a 64-bit sequence number

In Figure 1, the least-significant 32 bits of the sequence number are labeled the "Legacy Sequence Number". This is the portion of the sequence number that is stored in the Sequence Number field of the standard TCP header defined in [RFC0793]. In Figure 1, the most-significant 32 bits of the sequence number are labeled the "Sequence Number Extension". This is the portion of the sequence number that is stored in the 64-bit Sequence Number Option, which is defined in this document.

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

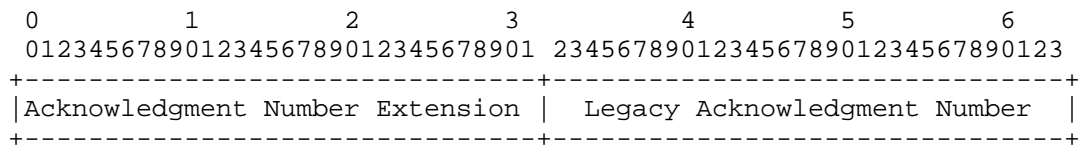


Figure 2: Logical division of a 64-bit acknowledgment number

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.

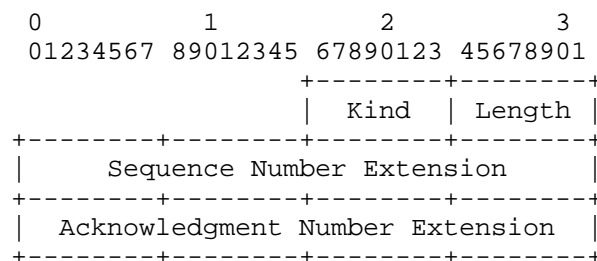


Figure 3: The 64-bit Sequence Number Option

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.

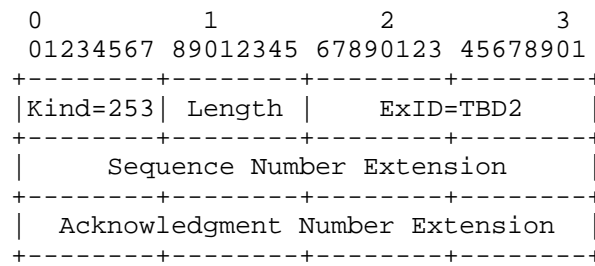


Figure 4: The 64-bit Sequence Number Option with ExID

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.

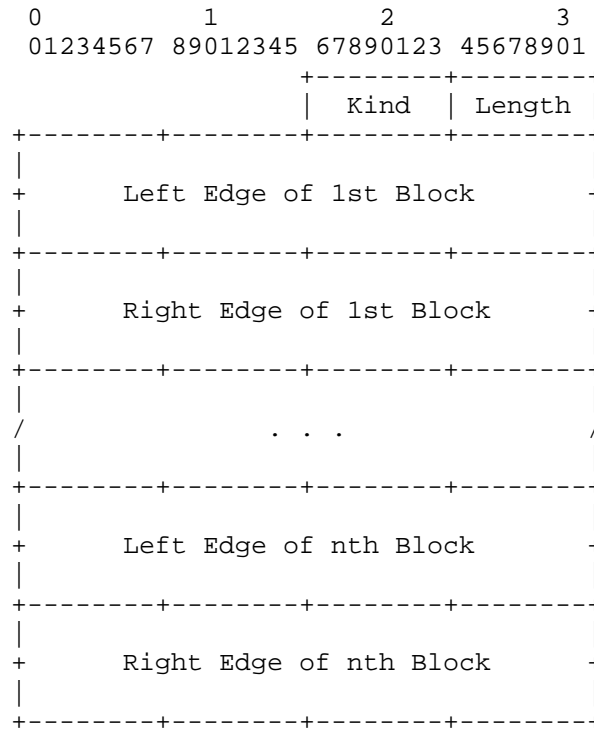


Figure 5: The 64-bit SACK Option

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.

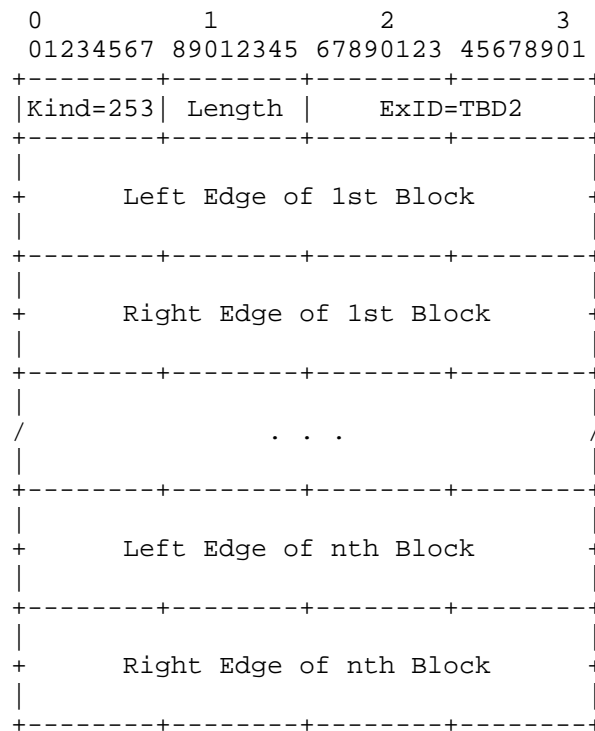


Figure 6: The 64-bit SACK Option with ExID

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
Obsoletes: 2140
Expires: July 2019

J. Touch
Independent Consultant
M. Welzl
S. Islam
University of Oslo
January 4, 2019

TCP Control Block Interdependence
draft-touch-tcpm-2140bis-06.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 4, 2019.

Copyright Notice

Copyright (c) 2019 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 updates and replaces RFC 2140's description of interdependent TCP control blocks, in which 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.....	13
11. Security Considerations.....	15
12. IANA Considerations.....	15
13. References.....	15
13.1. Normative References.....	15
13.2. Informative References.....	16

14. Acknowledgments.....	18
15. Change log.....	18
16. Appendix A: TCB sharing history.....	20
17. Appendix B: Options.....	20

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 [RFC2140]. 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 updates RFC 2140's discussion of TCB state sharing and provides a complete replacement for that document. This state sharing affects only TCB initialization [RFC2140] and thus 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 protocol for establishing the initial shared state about the connection; we include the endpoint numbers and components (timers, flags) required upon commencement that are later used to help maintain that state. 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 can be initialized using context from past connections as follows:

TEMPORAL SHARING - TCB Initialization

Cached TCB	New TCB
-----	-----
old_MMS_S	old_MMS_S or not cached
old_MMS_R	old_MMS_R or not cached
old_sendMSS	old_sendMSS
old_PMTU	old_PMTU
old_RTT	old_RTT
old_RTTvar	old_RTTvar
old_option	(option specific)
old_ssthresh	old_ssthresh
old_snd_cwnd	old_snd_cwnd

Sections 8 and 9 discuss compatibility issues and implications of sharing the specific information listed above. Section 10 gives an overview of known implementations.

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][RFC4821][RFC8201], 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.

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. RFC 7413 states, "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." [RFC 7413]. 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 can be shared.

TEMPORAL SHARING - Option info

Cached	New
-----	-----
old_TFO_Cookie	old_TFO_Cookie
old_TFO_Failure	old_TFO_Failure

TEMPORAL SHARING - Cache Updates

Cached TCB	Current TCB	when?	New Cached TCB
old_MMS_S	curr_ MMS_S	OPEN	curr MMS_S
old_MMS_R	curr_ MMS_R	OPEN	curr_MMS_R
old_sendMSS	curr_sendMSS	MSSopt	curr_sendMSS
old_PMTU	curr_PMTU	PMTUD	curr_PMTU
old_RTT	curr_RTT	CLOSE	merge(curr,old)
old_RTTvar	curr_RTTvar	CLOSE	merge(curr,old)
old_option	curr option	ESTAB	(depends on option)
old_ssthresh	curr_ssthresh	CLOSE	merge(curr,old)
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 [RFC8201] 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, where "merge" is some function that combines the values of "curr" and "old". 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.

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 simply copy congestion window or ssthresh information; instead, the new values can be a function (f) of the cumulative values and the number of connections (N).

ENSEMBLE SHARING - TCB Initialization

Cached TCB	New TCB

old_MMS_S	old_MMS_S
old_MMS_R	old_MMS_R
old_sendMSS	old_sendMSS
old_PMTU	old_PMTU
old_RTT	old_RTT
old_RTTvar	old_RTTvar
old ssthresh sum	f(old ssthresh sum, N)
old snd_cwnd sum	f(old snd cwnd sum, N)
old_option	(option-specific)

Sections 8 and 9 discuss compatibility issues and implications of sharing the specific information listed above.

The table below gives an overview of option-specific information that can be shared.

ENSEMBLE SHARING Option info

Cached	New
-----	-----
old_TFO_Cookie	old_TFO_Cookie
old_TFO_Failure	old_TFO_Failure

ENSEMBLE SHARING - Cache Updates

Cached TCB	Current TCB	when?	New Cached TCB
-----	-----	-----	-----
old_MMS_S	curr_MMS_S	OPEN	curr_MMS_S
old_MMS_R	curr_MMS_R	OPEN	curr_MMS_R
old_sendMSS	curr_sendMSS	MSSopt	curr_sendMSS
old_PMTU	curr_PMTU	PMTUD /PLPMTUD	curr_PMTU
old_RTT	curr_RTT	update	rtt_update(old,curr)
old_RTTvar	curr_RTTvar	update	rtt_update(old,curr)
old ssthresh	curr ssthresh	update	adjust sum as appropriate
old snd_cwnd	curr snd_cwnd	update	adjust sum as appropriate
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 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. 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. The algorithm described in [Bal2] adjusts the initial cwnd depending on the cwnd values of ongoing connections.

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.

The authors of [Hul2] recommend caching ssthresh for temporal sharing only when flows are long. Some studies suggest that sharing ssthresh between short flows can deteriorate the performance of individual connections [Hul2, Dul6], although this may benefit aggregate network performance.

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, especially when TCP Fast Open is used [RFC7413].

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 the MSS, RTT, and congestion window values. By avoiding the so-called, "slow-start restart," performance can be optimized [Hu01]. 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 the initial version of this document [RFC2140], this update'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. Updates to RFC 2140

This document updates the description of TCB sharing in RFC 2140 and its associated impact on existing and new connection state, providing a complete replacement for that document [RFC2140]. It clarifies the previous description and terminology and extends the mechanism to its impact on new protocols and mechanisms, including multipath TCP, fast open, PLPMTUD, NAT, and the TCP Authentication Option.

The detailed impact on TCB state addresses TCB parameters in greater detail, addressing RSS in both the send and receive direction, MSS and send-MSS separately, adds path MTU and ssthresh, and addresses the impact on TCP option state.

New sections have been added to address compatibility issues and implementation observations. The relation of this work to T/TCP has

been moved to an appendix discussion on history, partly to reflect the deprecation of that protocol.

Finally, this document updates and significantly expands the referenced literature.

12. Security Considerations

These presented implementation methods do not have additional ramifications for explicit attacks. They 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.

13. IANA Considerations

There are no IANA implications or requests in this document.

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

14. References

14.1. 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.

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC2140] Touch, J., "TCP Control Block Interdependence", RFC 2140, April 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.
- [RFC8201] McCann, J., Deering, S., Mogul, J., Hinden, R. (Ed.), "Path MTU Discovery for IP version 6," RFC 8201, Jul. 2017.

14.2. 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.
- [Du16] Dukkupati, N., Yuchung C., and Amin V., "Research Impacting the Practice of Congestion Control." ACM SIGCOMM CCR (editorial), on-line post, July, 2016.
- [Hu01] Hugues, A., Touch, J., Heidemann, J., "Issues in Slow-Start Restart After Idle", draft-hughes-restart-00 (expired), Dec., 2001.
- [Hu12] Hurtig, P., Brunstrom, A., "Enhanced metric caching for short TCP flows," 2012 IEEE International Conference on Communications (ICC), Ottawa, ON, 2012, pp. 1209-1213.

- [Bal2] Barik, R., Welzl, M., Ferlin, S., Alay, O., " LISA: A Linked Slow-Start Algorithm for MPTCP", IEEE ICC, Kuala Lumpur, Malaysia, 23-27 May 2016.
- [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., Sathiaselalan, A., Secchi, R., "Updating TCP to Support Rate-Limited Traffic", RFC 7661, Oct. 2015

15. 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. Earlier revisions of this work received funding from a collaborative research project between the University of Oslo and Huawei Technologies Co., Ltd. and were partly supported by USC/ISI's Postel Center.

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

16. Change log

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

06:

- Changed to update 2140, cite it normatively, and summarize the updates in a separate section

05:

- Fixed some TBDs.

04:

- Removed BCP-style recommendations and fixed some TBDs.

03:

- Updated Touch's affiliation and address information

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 MSS_S and MSS_R from RFC1122; fixed the use of MSS and MTU
- Added information about option sharing, listed options in the appendix

Authors' Addresses

Joe Touch

Manhattan Beach, CA 90266
USA

Phone: +1 (310) 560-0334
Email: touch@strayalpha.com

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

17. 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].

18. 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:

TFO cookie (if TFO succeeded in the past)

