Internet Engineering Task Force W. Eddy, Ed.

Internet-Draft MTI Systems

Obsoletes: 793, 879, 2873, 6093, 6429, August 6, 2020

6528, 6691 (if approved)

Updates: 5961, 1122 (if approved)

Intended status: Standards Track

Expires: February 7, 2021

Transmission Control Protocol (TCP) Specification

draft-ietf-tcpm-rfc793bis-18

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 RFC 1122,

and should be considered as a replacement for the portions of that

document dealing with TCP requirements.

Also, it updates RFC 5961 by adding a

small clarification in reset handling while in the SYN-RECEIVED

state.

RFC EDITOR NOTE: If approved for publication as an RFC, this should

be marked additionally as "STD: 7" and replace RFC 793 in that role.

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/.

Eddy Expires February 7, 2021 [Page 1]

Internet-Draft TCP Specification August 2020

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 February 7, 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.

This document may contain material from IETF Documents or IETF

Contributions published or made publicly available before November

10, 2008. The person(s) controlling the copyright in some of this

material may not have granted the IETF Trust the right to allow

modifications of such material outside the IETF Standards Process.

Without obtaining an adequate license from the person(s) controlling

the copyright in such materials, this document may not be modified

outside the IETF Standards Process, and derivative works of it may

not be created outside the IETF Standards Process, except to format

it for publication as an RFC or to translate it into languages other

than English.

Table of Contents

1. Purpose and Scope . . . . . . . . . . . . . . . . . . . . . . 3

2. Introduction . . . . . . . . . . . . . . . . . . . . . . . . 4

2.1. Key TCP Concepts . . . . . . . . . . . . . . . . . . . . 5

3. Functional Specification . . . . . . . . . . . . . . . . . . 6

3.1. Header Format . . . . . . . . . . . . . . . . . . . . . . 6

3.2. Terminology Overview . . . . . . . . . . . . . . . . . . 11

3.2.1. Key Connection State Variables . . . . . . . . . . . 11

3.2.2. State Machine Overview . . . . . . . . . . . . . . . 13

3.3. Sequence Numbers . . . . . . . . . . . . . . . . . . . . 16

3.4. Establishing a connection . . . . . . . . . . . . . . . . 23

3.5. Closing a Connection . . . . . . . . . . . . . . . . . . 29

3.5.1. Half-Closed Connections . . . . . . . . . . . . . . . 32

Eddy Expires February 7, 2021 [Page 2]

Internet-Draft TCP Specification August 2020

3.6. Segmentation . . . . . . . . . . . . . . . . . . . . . . 32

3.6.1. Maximum Segment Size Option . . . . . . . . . . . . . 34

3.6.2. Path MTU Discovery . . . . . . . . . . . . . . . . . 35

3.6.3. Interfaces with Variable MTU Values . . . . . . . . . 36

3.6.4. Nagle Algorithm . . . . . . . . . . . . . . . . . . . 36

3.6.5. IPv6 Jumbograms . . . . . . . . . . . . . . . . . . . 37

3.7. Data Communication . . . . . . . . . . . . . . . . . . . 37

3.7.1. Retransmission Timeout . . . . . . . . . . . . . . . 38

3.7.2. TCP Congestion Control . . . . . . . . . . . . . . . 38

3.7.3. TCP Connection Failures . . . . . . . . . . . . . . . 38

3.7.4. TCP Keep-Alives . . . . . . . . . . . . . . . . . . . 39

3.7.5. The Communication of Urgent Information . . . . . . . 40

3.7.6. Managing the Window . . . . . . . . . . . . . . . . . 41

3.8. Interfaces . . . . . . . . . . . . . . . . . . . . . . . 46

3.8.1. User/TCP Interface . . . . . . . . . . . . . . . . . 46

3.8.2. TCP/Lower-Level Interface . . . . . . . . . . . . . . 55

3.9. Event Processing . . . . . . . . . . . . . . . . . . . . 57

3.10. Glossary . . . . . . . . . . . . . . . . . . . . . . . . 82

4. Changes from RFC 793 . . . . . . . . . . . . . . . . . . . . 87

5. IANA Considerations . . . . . . . . . . . . . . . . . . . . . 92

6. Security and Privacy Considerations . . . . . . . . . . . . . 93

7. Acknowledgements . . . . . . . . . . . . . . . . . . . . . . 94

8. References . . . . . . . . . . . . . . . . . . . . . . . . . 95

8.1. Normative References . . . . . . . . . . . . . . . . . . 95

8.2. Informative References . . . . . . . . . . . . . . . . . 96

Appendix A. Other Implementation Notes . . . . . . . . . . . . . 100

A.1. IP Security Compartment and Precedence . . . . . . . . . 100

A.1.1. Precedence . . . . . . . . . . . . . . . . . . . . . 101

A.1.2. MLS Systems . . . . . . . . . . . . . . . . . . . . . 101

A.2. Sequence Number Validation . . . . . . . . . . . . . . . 102

A.3. Nagle Modification . . . . . . . . . . . . . . . . . . . 102

A.4. Low Water Mark Settings . . . . . . . . . . . . . . . . . 102

Appendix B. TCP Requirement Summary . . . . . . . . . . . . . . 103

Author's Address . . . . . . . . . . . . . . . . . . . . . . . . 106

1. Purpose and Scope

In 1981, RFC 793 [13] was released, documenting the Transmission

Control Protocol (TCP), and replacing earlier TCP specifications

that have 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 [42]. Over time, a

number of errata have been filled against RFC 793, as well as

deficiencies in security, performance, and many other aspects. The number

Eddy Expires February 7, 2021 [Page 3]

Internet-Draft TCP Specification August 2020

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 that have been made to the base TCP

functional specification and unify them into an update of [RFC 793].

Some companion documents are referenced for

important algorithms used by TCP (e.g., congestion control), but

have not been attempted to be 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 implementation must support in order to interoperate. Because 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.

Also, 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.

Multipath considerations are out of scope. The reader may refer to [RFC 8684].

2. Introduction

[RFC793] 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

[RFC793]. It does not replicate or attempt to update the introduction

and philosophy content in Sections 1 and 2 of [RFC793]. 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.

Eddy Expires February 7, 2021 [Page 4]

Internet-Draft TCP Specification August 2020

The "TCP Roadmap" [42] 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., [29]) 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 [40], but these are not

strictly required or discussed in this document.

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

2.1 Terms and Conventions

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 [4][11] 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.

2.1. Key TCP Concepts

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

applications.

The application byte-stream is conveyed over the network via TCP

segments, 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.

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.

Eddy Expires February 7, 2021 [Page 5]

Internet-Draft TCP Specification August 2020

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 [45]. Further

description of the motivations for developing TCP and its role in the

Internet protocol stack can be found in Section 2 of [13] 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] [12]. A TCP header follows the

internet header, supplying information specific to the TCP protocol.

This division allows for the existence of host level protocols other

than TCP. In early development of the Internet suite of protocols,

the IP header fields had been a part of TCP.

0 1 2 3

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Source Port | Destination Port |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Sequence Number |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Acknowledgment Number |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Data | |C|E|U|A|P|R|S|F| |

| Offset| Rsrvd |W|C|R|C|S|S|Y|I| Window |

| | |R|E|G|K|H|T|N|N| |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Checksum | Urgent Pointer |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Options | Padding |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Data |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Note that one tick mark represents one bit position.

Figure 1: TCP Header Format

The description of the fields as follows:

Source Port: 16 bits

The source port number.

Eddy Expires February 7, 2021 [Page 6]

Internet-Draft TCP Specification August 2020

Destination Port: 16 bits

The destination port number.

Sequence Number: 32 bits

The sequence number of the first data octet in this segment (except

when SYN 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: 4 bits

The number of 32 bit words in the TCP Header. This indicates where

the data begins. The TCP header (even one including options) is an

integral number of 32 bits long.

Rsrvd - Reserved: 4 bits

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 [49].

Control Bits: 8 bits (from left to right) of currently assigned control bits

CWR: Congestion Window Reduced (see [8])

ECE: ECN-Echo (see [8])

URG: Urgent Pointer field significant

ACK: Acknowledgment field significant

PSH: Push Function (see the Send Call description in

Section 3.8.1)

RST: Reset the connection

SYN: Synchronize sequence numbers

FIN: No more data from sender

Window: 16 bits

The number of data octets beginning with the one indicated in the

acknowledgment field that the sender of this segment is willing to

accept.

Eddy Expires February 7, 2021 [Page 7]

Internet-Draft TCP Specification August 2020

The window size MUST be treated as an unsigned number, or else

large window sizes will appear like negative windows and TCP will

now work (MUST-1). It is RECOMMENDED that implementations will

reserve 32-bit fields for the send and receive window sizes in the

connection record and do all window computations with 32 bits (REC-

1).

Checksum: 16 bits

The checksum field is the 16 bit one's complement of the one's

complement sum of all 16 bit words in the header and text. 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 conceptually prefixed to

the TCP header (Figure X). The pseudo header is 96 bits for IPv4 and 320 bits

for IPv6. This 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.

+--------+--------+--------+--------+

| Source Address |

+--------+--------+--------+--------+

| Destination Address |

+--------+--------+--------+--------+

| zero | PTCL | TCP Length |

+--------+--------+--------+--------+

Figure X: IPv4 TCP Pseudo-Header

Psuedo 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

Eddy Expires February 7, 2021 [Page 8]

Internet-Draft TCP Specification August 2020

TCP Length: the TCP header length plus the data length in octets

(this is not an explicitly transmitted quantity, but is

computed), and it does not count the 12 octets of the pseudo

header.

For IPv6, the pseudo header is contained in Section 8.1 of RFC 8200

[12], and contains the IPv6 Source Address and Destination Address,

an Upper Layer Packet Length (a 32-bit value otherwise equivalent

to TCP Length in the IPv4 pseudo header), three bytes of zero-

padding, and a Next Header value (differing from the IPv6 header

value in the case of extension headers present in between IPv6 and

TCP).

The TCP checksum is never optional. The sender MUST generate it

(MUST-2) and the receiver MUST check it (MUST-3).

Urgent Pointer: 16 bits

This field communicates the current value of the urgent pointer as

a positive offset from the sequence number in this segment. The

urgent pointer points to the sequence number of the octet following

the urgent data. This field is only be interpreted in segments

with the URG control bit set.

Options: variable

Options may occupy space at the end of the TCP header and are a

multiple of 8 bits in length. All options are included in the

checksum. An option may begin on any octet boundary. There are

two cases for the format of an option:

Case 1: A single octet of option-kind.

Case 2: An octet of option-kind (Kind), an octet of option-length, and

the actual option-data octets.

The option-length counts the two octets of option-kind and option-

length as well as the option-data octets.

Note that the list of options may be shorter than the data offset

field might imply. The content of the header beyond the End-of-

Option option must be header padding (i.e., zero).

The list of all currently defined options is managed by IANA [48],

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 [39].

Eddy Expires February 7, 2021 [Page 9]

Internet-Draft TCP Specification August 2020

A given TCP implementation can support any currently defined

options, but the following options MUST be supported (MUST-4) (kind

indicated in octal):

Kind Length Meaning

---- ------ -------

0 - End of option list.

1 - No-Operation.

2 4 Maximum Segment Size.

A TCP 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

have length fields). 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).

Specific Option Definitions

End of Option List

+--------+

|00000000|

+--------+

Kind=0

This option code indicates the end of the option list. This

might not coincide with the end of the TCP header according to

the Data Offset field. This is used at the end of all options,

not the end of each option, and need only be used if the end of

the options would not otherwise coincide with the end of the TCP

header.

No-Operation

+--------+

|00000001|

+--------+

Kind=1

This option code 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

Eddy Expires February 7, 2021 [Page 10]

Internet-Draft TCP Specification August 2020

receivers MUST be prepared to process options even if they do

not begin on a word boundary (MUST-64).

Maximum Segment Size (MSS)

+--------+--------+---------+--------+

|00000010|00000100| max seg size |

+--------+--------+---------+--------+

Kind=2 Length=4

Maximum Segment Size Option Data: 16 bits

If this option is present, then it communicates the maximum

receive segment size at the TCP 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 3.6.1.

Experimental TCP option values are defined in [22], and [39]

describes the current recommended usage for these experimental

values.

Note: There is an ongoing work to extend the space available for

TCP options, such as [53].

Padding: variable

Padding is used to ensure that the TCP header ends

and data begins on a 32 bit boundary. The padding is composed of

zeros.

3.2. Terminology Overview

This section includes an overview of key terms needed to understand

the detailed protocol operation in the rest of the document. There

is a traditional glossary of terms in Section 3.10.

3.2.1. 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 the remembering of several

variables. We conceive of these variables being stored in a

connection record called a Transmission Control Block or TCB. Among

the variables stored in the TCB are the local and remote IP addresses

Eddy Expires February 7, 2021 [Page 11]

Internet-Draft TCP Specification August 2020

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 2 3 4

----------|----------|----------|----------

SND.UNA SND.NXT SND.UNA

+SND.WND

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 2: Send Sequence Space

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

Figure 2.

Eddy Expires February 7, 2021 [Page 12]

Internet-Draft TCP Specification August 2020

1 2 3

----------|----------|----------

RCV.NXT RCV.NXT

+RCV.WND

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 3: Receive Sequence Space

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

Figure 3.

There are also some variables used frequently in the discussion that

take their values from the fields of the current segment.

Current Segment Variables:

SEG.SEQ - segment sequence number

SEG.ACK - segment acknowledgment number

SEG.LEN - segment length

SEG.WND - segment window

SEG.UP - segment urgent pointer

3.2.2. State Machine Overview

A connection progresses through a series of states during its

lifetime. The states are: LISTEN, SYN-SENT, SYN-RECEIVED,

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

TIME-WAIT, and the fictional state CLOSED. CLOSED is fictional

because it represents the state when there is no TCB, and therefore,

no connection. Briefly the meanings of the states are:

LISTEN - represents waiting for a connection request from any

remote TCP 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.

Eddy Expires February 7, 2021 [Page 13]

Internet-Draft TCP Specification August 2020

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.

CLOSED - represents no connection state at all.

A TCP connection progresses from one state to another in response to

events. The events are the user calls, OPEN, SEND, RECEIVE, CLOSE,

ABORT, and STATUS; the incoming segments, particularly those

containing the SYN, ACK, RST and FIN flags; and timeouts.

The state diagram in Figure 4 illustrates only state changes,

together with the causing events and resulting actions, but addresses

neither error conditions nor actions 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.

+---------+ ---------\ active OPEN

| CLOSED | \ -----------

+---------+<---------\ \ create TCB

| ^ \ \ snd SYN

passive OPEN | | CLOSE \ \

------------ | | ---------- \ \

create TCB | | delete TCB \ \

Eddy Expires February 7, 2021 [Page 14]

Internet-Draft TCP Specification August 2020

V | \ \

rcv RST (note 1) +---------+ CLOSE | \

-------------------->| LISTEN | ---------- | |

/ +---------+ delete TCB | |

/ rcv SYN | | SEND | |

/ ----------- | | ------- | V

+--------+ snd SYN,ACK / \ snd SYN +--------+

| |<----------------- ------------------>| |

| SYN | rcv SYN | SYN |

| RCVD |<-----------------------------------------------| SENT |

| | snd SYN,ACK | |

| |------------------ -------------------| |

+--------+ rcv ACK of SYN \ / rcv SYN,ACK +--------+

| -------------- | | -----------

| x | | snd ACK

| V V

| CLOSE +---------+

| ------- | ESTAB |

| snd FIN +---------+

| CLOSE | | rcv FIN

V ------- | | -------

+---------+ snd FIN / \ snd ACK +---------+

| FIN |<----------------- ------------------>| CLOSE |

| WAIT-1 |------------------ | WAIT |

+---------+ rcv FIN \ +---------+

| rcv ACK of FIN ------- | CLOSE |

| -------------- snd ACK | ------- |

V x V snd FIN V

+---------+ +---------+ +---------+

|FINWAIT-2| | CLOSING | | LAST-ACK|

+---------+ +---------+ +---------+

| rcv ACK of FIN | rcv ACK of FIN |

| rcv FIN -------------- | Timeout=2MSL -------------- |

| ------- x V ------------ x V

\ snd ACK +---------+delete TCB +---------+

------------------------>|TIME WAIT|------------------>| CLOSED |

+---------+ +---------+

Note 1: The transition from SYN-RECEIVED to LISTEN on receiving a RST is

conditional on having reached SYN-RECEIVED after a passive open.

Note 2: An unshown transition exists from FIN-WAIT-1 to TIME-WAIT if

a FIN is received and the local FIN is also acknowledged.

Figure 4: TCP Connection State Diagram

Eddy Expires February 7, 2021 [Page 15]

Internet-Draft TCP Specification August 2020

3.3. Sequence Numbers

A fundamental notion in the design is that every octet of data sent

over a TCP connection has a sequence number. Since every octet is

sequenced, each of them can be acknowledged. The acknowledgment

mechanism employed is cumulative so that an acknowledgment of

sequence number X indicates that all octets up to but not including X

have been received. This mechanism allows for straight-forward

duplicate detection in the presence of retransmission. Numbering of

octets within a segment is that the first data octet immediately

following the header is the lowest numbered, and the following octets

are numbered consecutively.

It is essential to remember that the actual sequence number space is

finite, though very large. This space ranges from 0 to 2\*\*32 - 1.

Since the space is finite, all arithmetic dealing with sequence

numbers must be performed modulo 2\*\*32. This unsigned arithmetic

preserves the relationship of sequence numbers as they cycle from

2\*\*32 - 1 to 0 again. There are some subtleties to computer modulo

arithmetic, so great care should be taken in programming the

comparison of such values. The symbol "=<" means "less than or

equal" (modulo 2\*\*32).

The typical kinds of sequence number comparisons 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)

Eddy Expires February 7, 2021 [Page 16]

Internet-Draft TCP Specification August 2020

SEG.SEQ = first sequence number of a segment

SEG.LEN = the number of octets occupied by the data in the segment

(counting SYN and FIN)

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

A new acknowledgment (called an "acceptable ack"), is one for which

the inequality below holds:

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

A segment on the retransmission queue is fully acknowledged if the

sum of its sequence number and length is less or equal than the

acknowledgment value in the incoming segment.

When data is received the following comparisons are needed:

RCV.NXT = next sequence number expected on an incoming segments,

and is the left or lower edge of the receive window

RCV.NXT+RCV.WND-1 = last sequence number expected on an incoming

segment, and is the right or upper edge of the receive window

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

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

segment

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

space if

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

or

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

The first part of this test checks to see if the beginning of the

segment falls in the window, the second part of the test checks to

see if the end of the segment falls in the window; if the segment

passes either part of the test it contains data in the window.

Actually, it is a little more complicated than this. Due to zero

windows and zero length segments, we have four cases for the

acceptability of an incoming segment:

Eddy Expires February 7, 2021 [Page 17]

Internet-Draft TCP Specification August 2020

Segment Receive Test

Length Window

------- ------- -------------------------------------------

0 0 SEG.SEQ = RCV.NXT

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

>0 0 not acceptable

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

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

Note that when the receive window is zero no segments should be

acceptable except ACK segments. Thus, it is be 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.

Initial Sequence Number Selection

The protocol places no restriction on a particular connection being

used over and over again. A connection is defined by a pair of

sockets. New instances of a connection will be referred to as

incarnations of the connection. The problem that arises from this is

-- "how does the TCP 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.

Eddy Expires February 7, 2021 [Page 18]

Internet-Draft TCP Specification August 2020

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.

The recommended ISN generator is based on the combination of a

(possibly fictitious) 32 bit clock whose low order bit is incremented

roughly every 4 microseconds, and a pseudorandom hash function (PRF).

The clock component is intended to insure that with a Maximum Segment

Lifetime (MSL), generated ISNs will be unique, since it cycles

approximately every 4.55 hours, which is much longer than the MSL.

This recommended algorithm is further described in RFC 6528 [36] and

builds on the basic clock-driven algorithm from RFC 793.

A TCP implementation MUST use a clock-driven selection of initial

sequence numbers (MUST-8), and SHOULD generate its Initial Sequence

Numbers with the expression:

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

where M is the 4 microsecond timer, and F() is a pseudorandom

function (PRF) of the connection's identifying parameters ("localip,

localport, remoteip, remoteport") and a secret key ("secretkey")

(SHLD-1). F() MUST NOT be computable from the outside (MUST-9), or

an attacker could still guess at sequence numbers from the ISN used

for some other connection. The PRF could be implemented as a

cryptographic 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 [36].

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 ISN's.

Eddy Expires February 7, 2021 [Page 19]

Internet-Draft TCP Specification August 2020

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 (3WHC).

A 3WHC 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 ISN's. The receiver of the

first SYN has no way of knowing whether the segment was an old

delayed one or not, unless it remembers the last sequence number used

on the connection (which is not always possible), and so it must ask

the sender to verify this SYN. The three way handshake and the

advantages of a clock-driven scheme are discussed in [55].

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 have been 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

Eddy Expires February 7, 2021 [Page 20]

Internet-Draft TCP Specification August 2020

it need not wait at all; it must only be sure to use sequence numbers

larger than those recently used.

The TCP Quiet Time Concept

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 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 very large to reduce the probability that a wandering

duplicate will cause trouble upon arrival. At 2 megabits/sec. it

takes 4.5 hours to use up 2\*\*32 octets of sequence space. Since the

maximum segment lifetime in the net is not likely to exceed a few

tens of seconds, this is deemed ample protection for foreseeable

nets, even if data rates escalate to l0's of megabits/sec. At 100

megabits/sec, the cycle time is 5.4 minutes, which may be a little

short, but still within reason.

The basic duplicate detection and sequencing algorithm in TCP can be

defeated, however, if a source TCP 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 re-

form an earlier connection (possibly after half-open connection

Eddy Expires February 7, 2021 [Page 21]

Internet-Draft TCP Specification August 2020

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 S1, 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 S1 = ISN(t)

-- last used sequence number on old incarnation of connection! If

the recovery occurs quickly enough, any old duplicates in the net

bearing sequence numbers in the neighborhood of S1 may arrive and be

treated as new packets by the receiver of the new incarnation of the

connection.

The problem is that the recovering host may not know for how long it

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 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

the last emitted segment, if a new connection is started too soon and

uses any of the sequence numbers in the space-time footprint of the

Eddy Expires February 7, 2021 [Page 22]

Internet-Draft TCP Specification August 2020

last segment of the previous connection incarnation, there is a

potential sequence number overlap area that could cause confusion at

the receiver.

3.4. Establishing a connection

The "three-way handshake" is the procedure used to establish a

connection. This procedure normally is initiated by one TCP 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 the

implementation of a trade-off between memory and messages to provide

information for this checking.

The simplest 3WHC is shown in Figure 5.

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.

Eddy Expires February 7, 2021 [Page 23]

Internet-Draft TCP Specification August 2020

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 5: Basic 3-Way Handshake for Connection Synchronization

In line 2 of Figure 5, 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 ACK's!).

Simultaneous initiation is only slightly more complex, as is shown in

Figure 6. 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 6: Simultaneous Connection Synchronization

Eddy Expires February 7, 2021 [Page 24]

Internet-Draft TCP Specification August 2020

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 7: Recovery from Old Duplicate SYN

As a simple example of recovery from old duplicates, consider

Figure 7. At line 3, an old duplicate SYN arrives at TCP 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.

Eddy Expires February 7, 2021 [Page 25]

Internet-Draft TCP Specification August 2020

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 8. 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 8: Half-Open Connection Discovery

Eddy Expires February 7, 2021 [Page 26]

Internet-Draft TCP Specification August 2020

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 5.

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 9. 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 9: Active Side Causes Half-Open Connection Discovery

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

Eddy Expires February 7, 2021 [Page 27]

Internet-Draft TCP Specification August 2020

A variety of other cases are possible, all of which are accounted for

by the following rules for RST generation and processing.

Reset Generation

As a general rule, reset (RST) 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 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

Eddy Expires February 7, 2021 [Page 28]

Internet-Draft TCP Specification August 2020

state. The reset takes its sequence number from the ACK field of

the incoming segment.

Reset Processing

In all states except SYN-SENT, all reset (RST) segments are validated

by checking their SEQ-fields. A reset is valid if its sequence

number is in the window. In the SYN-SENT state (a RST received in

response to an initial SYN), the RST is acceptable if the ACK field

acknowledges the SYN.

The receiver of a RST first validates it, then changes state. If the

receiver was in the LISTEN state, it ignores it. If the receiver was

in SYN-RECEIVED state and had previously been in the LISTEN state,

then the receiver returns to the LISTEN state, otherwise the receiver

aborts the connection and goes to the CLOSED state. If the receiver

was in any other state, it aborts the connection and advises the user

and goes to the CLOSED state.

TCP implementations SHOULD allow a received RST segment to include

data (SHLD-2).

3.5. Closing a Connection

CLOSE is an operation meaning "I have no more data to send." The

notion of closing a full-duplex connection is subject to ambiguous

interpretation, of course, since it may not be obvious how to treat

the receiving side of the connection. We have chosen to treat CLOSE

in a simplex fashion. The user who CLOSEs may continue to RECEIVE

until 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 his 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

Eddy Expires February 7, 2021 [Page 29]

Internet-Draft TCP Specification August 2020

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

signal

3) Both users CLOSE simultaneously

Case 1: Local user initiates the close

In this case, a FIN segment can be constructed and placed on the

outgoing segment queue. No further SENDs from the user will be

accepted by the TCP 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. 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.

Eddy Expires February 7, 2021 [Page 30]

Internet-Draft TCP Specification August 2020

TCP Peer A TCP Peer B

1. ESTABLISHED ESTABLISHED

2. (Close)

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

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

4. (Close)

TIME-WAIT <-- <SEQ=300><ACK=101><CTL=FIN,ACK> <-- LAST-ACK

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

6. (2 MSL)

CLOSED

Figure 11: Normal Close Sequence

TCP Peer A TCP Peer B

1. ESTABLISHED ESTABLISHED

2. (Close) (Close)

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

<-- <SEQ=300><ACK=100><CTL=FIN,ACK> <--

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

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

<-- <SEQ=301><ACK=101><CTL=ACK> <--

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

4. TIME-WAIT TIME-WAIT

(2 MSL) (2 MSL)

CLOSED CLOSED

Figure 12: Simultaneous Close Sequence

A TCP connection may terminate in two ways: (1) the normal TCP close

sequence using a FIN handshake, and (2) an "abort" in which one or

more RST segments are sent and the connection state is immediately

discarded. If the local TCP connection is closed by the remote side

due to a FIN or RST received from the remote side, then the local

application MUST be informed whether it closed normally or was

aborted (MUST-12).

Eddy Expires February 7, 2021 [Page 31]

Internet-Draft TCP Specification August 2020

3.5.1. Half-Closed Connections

The normal TCP close sequence delivers buffered data reliably in both

directions. Since the two directions of a TCP connection are closed

independently, it is possible for a connection to be "half closed,"

i.e., closed in only one direction, and a host is permitted to

continue sending data in the open direction on a half-closed

connection.

A host MAY implement a "half-duplex" TCP close sequence, so that an

application that has called CLOSE cannot continue to read data from

the connection (MAY-1). If such a host issues a CLOSE call while

received data is still pending in 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 [18] section 2.17 for

discussion.

When a connection is closed actively, it MUST linger in TIME-WAIT

state for a time 2xMSL (Maximum Segment Lifetime) (MUST-13).

However, it MAY accept a new SYN from the remote TCP 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 [34] in order to support higher connection

establishment rates. This algorithm for reducing TIME-WAIT is a Best

Current Practice that SHOULD be implemented, since timestamp options

are commonly used, and using them to reduce TIME-WAIT provides

benefits for busy Internet servers (SHLD-4).

3.6. Segmentation

The term "segmentation" refers to the activity TCP performs when

ingesting a stream of bytes from a sending application and

packetizing that stream of bytes into TCP segments. Individual TCP

segments often do not correspond one-for-one to individual send (or

socket write) calls from the application. Applications may perform

writes at the granularity of messages in the upper layer protocol,

but TCP guarantees no boundary coherence between the TCP segments

sent and received versus user application data read or write buffer

boundaries. In some specific protocols, such as RDMA using DDP and

MPA [26], there are performance optimizations possible when the

Eddy Expires February 7, 2021 [Page 32]

Internet-Draft TCP Specification August 2020

relation between TCP segments and application data units can be

controlled, and MPA includes a specific mechanism for detecting and

verifying this relationship between TCP segments and application

message data structures, but this is specific to applications like

RDMA. In general, multiple goals influence the sizing of TCP

segments created by a TCP implementation.

Goals driving the sending of larger segments include:

o Reducing the number of packets in flight within the network.

o Increasing processing efficiency and potential performance by

enabling a smaller number of interrupts and inter-layer

interactions.

o Limiting the overhead of TCP headers.

Note that the performance benefits of sending larger segments may

decrease as the size increases, and there may be boundaries where

advantages are reversed. For instance, on some 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:

o 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.

o Preventing delays to the application data stream, especially when

TCP is waiting on the application to generate more data, or when

the application is waiting on an event or input from its peer in

order to generate more data.

o Enabling "fate sharing" between TCP segments and lower-layer data

units (e.g., below IP, for links with cell or frame sizes smaller

than the IP MTU).

Towards meeting these competing sets of goals, TCP includes several

mechanisms, including the Maximum Segment Size option, Path MTU

Discovery, the Nagle algorithm, and support for IPv6 Jumbograms, as

discussed in the following subsections.

Eddy Expires February 7, 2021 [Page 33]

Internet-Draft TCP Specification August 2020

3.6.1. 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 [15]) and the largest transmission size permitted by

the IP layer (EMTU\_S [15]):

Eff.snd.MSS =

min(SendMSS+20, MMS\_S) - TCPhdrsize - IPoptionsize

where:

o SendMSS is the MSS value received from the remote host, or the

default 536 for IPv4 or 1220 for IPv6, if no MSS option is

received.

o MMS\_S is the maximum size for a transport-layer message that TCP

may send.

o TCPhdrsize is the size of the fixed TCP header and any options.

This is 20 in the (rare) case that no options are present, but may

be larger if TCP options are to be sent. Note that some options

may not be included on all segments, but that for each segment

sent, the sender should adjust the data length accordingly, within

the Eff.snd.MSS.

o IPoptionsize is the size of any IP options associated with a TCP

connection. Note that some options may not be included on all

packets, but that for each segment sent, the sender should adjust

the data length accordingly, within the Eff.snd.MSS.

The MSS value to be sent in an MSS option should be equal to the

effective MTU minus the fixed IP and TCP headers. By ignoring both

IP and TCP options when calculating the value for the MSS option, if

there are any IP or TCP options to be sent in a packet, then the

Eddy Expires February 7, 2021 [Page 34]

Internet-Draft TCP Specification August 2020

sender must decrease the size of the TCP data accordingly. RFC 6691

[37] 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 [15].

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.

3.6.2. Path MTU Discovery

A TCP implementation may be aware of the MTU on directly connected

links, but will rarely have insight about MTUs across an entire

network path. For IPv4, RFC 1122 provides an IP-layer recommendation

on the default effective MTU for sending to be less than or equal to

576 for destinations not directly connected. For IPv6, this would be

1280. In all cases, however, implementation of Path MTU Discovery

(PMTUD) and Packetization Layer Path MTU Discovery (PLPMTUD) is

strongly recommended in order for TCP to improve segmentation

decisions. Both PMTUD and PLPMTUD help TCP choose segment sizes that

avoid both on-path (for IPv4) and source fragmentation (IPv4 and

IPv6).

PMTUD for IPv4 [2] or IPv6 [3] is implemented in conjunction between

TCP, IP, and ICMP protocols. It relies both on avoiding source

fragmentation and setting the IPv4 DF (don't fragment) flag, the

latter to inhibit on-path fragmentation. It relies on ICMP errors

from routers along the path, whenever a segment is too large to

traverse a link. Several adjustments to a TCP implementation with

PMTUD are described in RFC 2923 in order to deal with problems

experienced in practice [7]. PLPMTUD [23] 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.

Eddy Expires February 7, 2021 [Page 35]

Internet-Draft TCP Specification August 2020

The TCP MSS option specifies an upper bound for the size of packets

that can be received. Hence, setting the value in the MSS option too

small can impact the ability for PMTUD or PLPMTUD to find a larger

path MTU. RFC 1191 discusses this implication of many older TCP

implementations setting MSS to 536 for non-local destinations, rather

than deriving it from the MTUs of connected interfaces as

recommended.

3.6.3. Interfaces with Variable MTU Values

The effective MTU can sometimes vary, as when used with variable

compression, e.g., RObust Header Compression (ROHC) [30]. It is

tempting for a TCP implementation to want to advertise the largest

possible MSS, to support the most efficient use of compressed

payloads. Unfortunately, some compression schemes occasionally need

to transmit full headers (and thus smaller payloads) to resynchronize

state at their endpoint compressors/decompressors. If the largest

MTU is used to calculate the value to advertise in the MSS option,

TCP retransmission may interfere with compressor resynchronization.

As a result, when the effective MTU of an interface varies 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).

3.6.4. Nagle Algorithm

The "Nagle algorithm" was described in RFC 896 [14] and was

recommended in RFC 1122 [15] 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 [29].

Eddy Expires February 7, 2021 [Page 36]

Internet-Draft TCP Specification August 2020

3.6.5. IPv6 Jumbograms

In order to support TCP over IPv6 Jumbograms, implementations need to

be able to send TCP segments larger than the 64KB limit that the MSS

option can convey. RFC 2675 [6] defines that an MSS value of 65,535

bytes is to be treated as infinity, and Path MTU Discovery [3] is

used to determine the actual MSS.

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 [6], and the present IPv6 Node Requiements does not

include support for Jumbograms [47].

3.7. 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, as does the FIN control

flag in an incoming segment.

Eddy Expires February 7, 2021 [Page 37]

Internet-Draft TCP Specification August 2020

3.7.1. Retransmission Timeout

Because of the variability of the networks that compose an

internetwork system and the wide range of uses of TCP connections the

retransmission timeout (RTO) must be dynamically determined.

The RTO MUST be computed according to the algorithm in [9], including

Karn's algorithm for taking RTT samples (MUST-18).

RFC 793 contains an early example procedure for computing the RTO.

This was then replaced by the algorithm described in RFC 1122, and

subsequently updated in RFC 2988, and then again in RFC 6298.

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

[38]. 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.

3.7.2. TCP Congestion Control

RFC 1122 required implementation of Van Jacobson's congestion control

algorithm combining slow start with congestion avoidance. RFC 2581

provided IETF Standards Track description of this, along with fast

retransmit and fast recovery. RFC 5681 is the current description of

these algorithms and is the current standard for TCP congestion

control.

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

Explicit Congestion Notification (ECN) was defined in RFC 3168 and is

an IETF Standards Track enhancement that has many benefits [44].

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

8).

3.7.3. 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):

Eddy Expires February 7, 2021 [Page 38]

Internet-Draft TCP Specification August 2020

(a) There are two thresholds R1 and R2 measuring the amount of

retransmission that has occurred for the same segment. R1 and R2

might be measured in time units or as a count of retransmissions.

(b) When the number of transmissions of the same segment reaches

or exceeds threshold R1, pass negative advice (see

Section 3.3.1.4 of [15]) 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 (User

Telnet) application program to inform the user, for example.

The value of R1 SHOULD correspond to at least 3 retransmissions, at

the current RTO (SHLD-10). The value of R2 SHOULD correspond to at

least 100 seconds (SHLD-11).

An attempt to open a TCP connection could fail with excessive

retransmissions of the SYN segment or by receipt of a RST segment or

an ICMP Port Unreachable. SYN retransmissions MUST be handled in the

general way just described for data retransmissions, including

notification of the application layer.

However, the values of R1 and R2 may be different for SYN and data

segments. In particular, R2 for a SYN segment MUST be set large

enough to provide retransmission of the segment for at least 3

minutes (MUST-23). The application can close the connection (i.e.,

give up on the open attempt) sooner, of course.

3.7.4. TCP Keep-Alives

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

Eddy Expires February 7, 2021 [Page 39]

Internet-Draft TCP Specification August 2020

keep-alives are included, the application MUST be able to turn them

on or off for each TCP connection (MUST-24), and they MUST default to

off (MUST-25).

Keep-alive packets MUST only be sent when no data or acknowledgement

packets have been received for the connection within an interval

(MUST-26). This interval MUST be configurable (MUST-27) and MUST

default to no less than two hours (MUST-28).

It is extremely important to remember that ACK segments that contain

no data are not reliably transmitted by TCP. Consequently, if a

keep-alive mechanism is implemented it MUST NOT interpret failure to

respond to any specific probe as a dead connection (MUST-29).

An implementation SHOULD send a keep-alive segment with no data

(SHLD-12); however, it MAY be configurable to send a keep-alive

segment containing one garbage octet (MAY-6), for compatibility with

erroneous TCP implementations.

3.7.5. The Communication of Urgent Information

As a result of implementation differences and middlebox interactions,

new applications SHOULD NOT employ the TCP urgent mechanism (SHLD-

13). However, TCP implementations MUST still include support for the

urgent mechanism (MUST-30). Details can be found in RFC 6093 [33].

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.

Eddy Expires February 7, 2021 [Page 40]

Internet-Draft TCP Specification August 2020

To send an urgent indication the user must also send at least one

data octet. If the sending user also indicates a push, timely

delivery of the urgent information to the destination process is

enhanced.

A TCP implementation MUST support a sequence of urgent data of any

length (MUST-31). [15]

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. There MUST (MUST-33) be a way for the

application to learn how much urgent data remains to be read from the

connection, or at least to determine whether or not more urgent data

remains to be read [15].

3.7.6. 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.

Eddy Expires February 7, 2021 [Page 41]

Internet-Draft TCP Specification August 2020

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 [15]

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 "useable

window" (see Section 3.7.6.2.1) 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).

3.7.6.1. Zero Window Probing

The sending TCP peer must be prepared to accept from the user and

send at least one octet of new data even if the send window is zero.

The sending TCP peer must regularly retransmit to the receiving TCP

peer even when the window is zero, in order to "probe" the window.

Two minutes is recommended for the retransmission interval when the

window is zero. This retransmission is essential to guarantee that

when either TCP 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 RFC1122. The

behavior is subject to the implementation's resource management

concerns, as noted in [35].

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).

Eddy Expires February 7, 2021 [Page 42]

Internet-Draft TCP Specification August 2020

The transmitting host SHOULD send the first zero-window probe when a

zero window has existed for the retransmission timeout period (SHLD-

29) (Section 3.7.1), and SHOULD increase exponentially the

interval between successive probes (SHLD-30).

3.7.6.2. 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.

3.7.6.2.1. 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 3.6.4 additionally describes how to

coalesce short segments.

The sender's SWS avoidance algorithm is more difficult than the

receivers's, because the sender does not know (directly) the

receiver's total buffer space RCV.BUFF. An approach that has been

found to work well is for the sender to calculate Max(SND.WND), the

maximum send window it has seen so far on the connection, and to use

this value as an estimate of RCV.BUFF. Unfortunately, this can only

be an estimate; the receiver may at any time reduce the size of

RCV.BUFF. To avoid a resulting deadlock, it is necessary to have a

timeout to force transmission of data, overriding the SWS avoidance

algorithm. In practice, this timeout should seldom occur.

The "useable window" is:

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

i.e., the offered window less the amount of data sent but not

acknowledged. If D is the amount of data queued in the sending TCP

endpoint but not yet sent, then the following set of rules is

recommended.

Send data:

Eddy Expires February 7, 2021 [Page 43]

Internet-Draft TCP Specification August 2020

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

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

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

i.e., if:

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

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

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

i.e., if:

[SND.NXT = SND.UNA and]

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

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

Here Fs is a fraction whose recommended value is 1/2. The override

timeout should be in the range 0.1 - 1.0 seconds. It may be

convenient to combine this timer with the timer used to probe zero

windows (Section 3.7.6.1).

3.7.6.2.2. 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 3.7.6.3) to determine when an ACK segment containing the

current window will really be sent to the receiver.

The solution to receiver SWS is to avoid advancing the right window

edge RCV.NXT+RCV.WND in small increments, even if data is received

from the network in small segments.

Suppose the total receive buffer space is RCV.BUFF. At any given

moment, RCV.USER octets of this total may be tied up with data that

has been received and acknowledged but that the user process has not

yet consumed. When the connection is quiescent, RCV.WND = RCV.BUFF

and RCV.USER = 0.

Keeping the right window edge fixed as data arrives and is

acknowledged requires that the receiver offer less than its full

Eddy Expires February 7, 2021 [Page 44]

Internet-Draft TCP Specification August 2020

buffer space, i.e., the receiver must specify a RCV.WND that keeps

RCV.NXT+RCV.WND constant as RCV.NXT increases. Thus, the total

buffer space RCV.BUFF is generally divided into three parts:

|<------- RCV.BUFF ---------------->|

1 2 3

----|---------|------------------|------|----

RCV.NXT ^

(Fixed)

1 - RCV.USER = data received but not yet consumed;

2 - RCV.WND = space advertised to sender;

3 - Reduction = space available but not yet

advertised.

The suggested SWS avoidance algorithm for the receiver is to keep

RCV.NXT+RCV.WND fixed until the reduction satisfies:

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

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

where Fr is a fraction whose recommended value is 1/2, and

Eff.snd.MSS is the effective send MSS for the connection (see

Section 3.6.1). When the inequality is satisfied, RCV.WND is set to

RCV.BUFF-RCV.USER.

Note that the general effect of this algorithm is to advance RCV.WND

in increments of Eff.snd.MSS (for realistic receive buffers:

Eff.snd.MSS < RCV.BUFF/2). Note also that the receiver must use its

own Eff.snd.MSS, assuming it is the same as the sender's.

3.7.6.3. 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), and in a stream of full-sized

segments there SHOULD be an ACK for at least every second segment

(SHLD-19). Excessive delays on ACK's can disturb the round-trip

timing and packet "clocking" algorithms. More complete discussion of

delayed ACK behavior is in Section 4.2 of RFC 5681 [29], including

Eddy Expires February 7, 2021 [Page 45]

Internet-Draft TCP Specification August 2020

rules for streams of segments that are not full-sized. Note that

there are several current practices that further lead to a reduced

number of ACKs, including generic receive offload (GRO), ACK

compression, and ACK decimation [20].

3.8. 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.

3.8.1. 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 [46] also identifies primitives provided by TCP, and

could be used as an additional reference for implementers.

TCP User Commands

The following sections functionally characterize a USER/TCP

interface. The notation used is similar to most procedure or

function calls in high level languages, but this usage is not

meant to rule out trap type service calls.

The user commands described below specify the basic functions the

TCP 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:

Eddy Expires February 7, 2021 [Page 46]

Internet-Draft TCP Specification August 2020

(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.

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 users authority to open a connection

with the specified DiffServ field value or security/

compartment. The absence of a DiffServ field value or

security/compartment specification in the OPEN call indicates

the default values must be used.

Eddy Expires February 7, 2021 [Page 47]

Internet-Draft TCP Specification August 2020

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 is used that was used in those previous

segments (MUST-45).

A TCP implementation MUST reject as an error a local OPEN call

for an invalid remote IP address (e.g., a broadcast or

multicast address) (MUST-46).

Send

Eddy Expires February 7, 2021 [Page 48]

Internet-Draft TCP Specification August 2020

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. 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).

The PSH bit is not a record marker and is independent of

segment boundaries. The transmitter SHOULD collapse successive

bits when it packetizes data, to send the largest possible

segment (SHLD-27).

If the PUSH flag is not set, the data may be combined with data

from subsequent SENDs for transmission efficiency. Note that

when the Nagle algorithm is in use, TCP implementations may

buffer the data before sending, without regard to the PUSH flag

(see Section 3.6.4).

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 3.7.6.2.1).

New applications SHOULD NOT set the URGENT flag [33] due to

implementation differences and middlebox issues (SHLD-13).

Eddy Expires February 7, 2021 [Page 49]

Internet-Draft TCP Specification August 2020

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

Eddy Expires February 7, 2021 [Page 50]

Internet-Draft TCP Specification August 2020

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.

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

Eddy Expires February 7, 2021 [Page 51]

Internet-Draft TCP Specification August 2020

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.

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 his 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.

Eddy Expires February 7, 2021 [Page 52]

Internet-Draft TCP Specification August 2020

Close also implies push function.

Status

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

This is an implementation dependent user command and could be

excluded without adverse effect. Information returned would

typically come from the TCB associated with the connection.

This command returns a data block containing the following

information:

local socket,

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/compartment,

and transmission timeout.

Depending on the state of the connection, or on the

implementation itself, some of this information may not be

available or meaningful. If the calling process is not

authorized to use this connection, an error is returned. This

prevents unauthorized processes from gaining information about

a connection.

Abort

Format: ABORT (local connection name)

This command causes all pending SENDs and RECEIVES to be

aborted, the TCB to be removed, and a special RESET message to

be sent to the 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.

Flush

Some TCP implementations have included a FLUSH call, which will

empty the TCP send queue of any data that the user has issued

Eddy Expires February 7, 2021 [Page 53]

Internet-Draft TCP Specification August 2020

SEND calls 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).

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 3.8.2.2 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 3.7.3)

\* Urgent pointer advance (see Section 3.7.5)

However, an application program that does not want to receive

such ERROR\_REPORT calls SHOULD be able to effectively disable

these calls (SHLD-20).

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

Eddy Expires February 7, 2021 [Page 54]

Internet-Draft TCP Specification August 2020

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).

3.8.2. 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 [12].

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) [43]. 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.

Eddy Expires February 7, 2021 [Page 55]

Internet-Draft TCP Specification August 2020

When received options are passed up to TCP from the IP layer, TCP

implementations 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.

3.8.2.1. 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).

3.8.2.2. 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.

[27] 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 [10] for discussion.

Soft Errors

For ICMP these include: Destination Unreachable -- codes 0, 1, 5,

Time Exceeded -- codes 0, 1, and Parameter Problem.

Eddy Expires February 7, 2021 [Page 56]

Internet-Draft TCP Specification August 2020

For ICMPv6 these include: Destination Unreachable -- codes 0 and 3,

Time Exceeded -- codes 0, 1, and Parameter Problem -- codes 0, 1, 2

Since these Unreachable messages indicate soft error conditions,

TCP 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). [27] 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 [27] section 4 describes widespread implementation behavior

that treats soft errors as hard errors during connection

establishment.

3.8.2.3. 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

[15]).

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.

3.9. 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.

Eddy Expires February 7, 2021 [Page 57]

Internet-Draft TCP Specification August 2020

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.

Eddy Expires February 7, 2021 [Page 58]

Internet-Draft TCP Specification August 2020

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/compartment, 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: precedence not allowed" or "error:

security/compartment 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 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 Foreign socket was not specified,

then return "error: remote socket unspecified".

Eddy Expires February 7, 2021 [Page 59]

Internet-Draft TCP Specification August 2020

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".

Eddy Expires February 7, 2021 [Page 60]

Internet-Draft TCP Specification August 2020

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

Foreign 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.

Eddy Expires February 7, 2021 [Page 61]

Internet-Draft TCP Specification August 2020

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 text already on hand, but not yet delivered to the

user. If no text is awaiting delivery, the RECEIVE will get a

"error: connection closing" response. Otherwise, any remaining

text can be used to satisfy the RECEIVE.

CLOSING STATE

LAST-ACK STATE

Eddy Expires February 7, 2021 [Page 62]

Internet-Draft TCP Specification August 2020

TIME-WAIT STATE

Return "error: connection closing".

Eddy Expires February 7, 2021 [Page 63]

Internet-Draft TCP Specification August 2020

CLOSE Call

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

If the user does not have access to such a connection, return

"error: connection illegal for this process".

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

LISTEN STATE

Any outstanding RECEIVEs are returned with "error: closing"

responses. Delete TCB, enter CLOSED state, and return.

SYN-SENT STATE

Delete the TCB and return "error: closing" responses to any

queued SENDs, or RECEIVEs.

SYN-RECEIVED STATE

If no SENDs have been issued and there is no pending data to

send, then form a FIN segment and send it, and enter FIN-WAIT-1

state; otherwise queue for processing after entering

ESTABLISHED state.

ESTABLISHED STATE

Queue this until all preceding SENDs have been segmentized,

then form a FIN segment and send it. In any case, enter FIN-

WAIT-1 state.

FIN-WAIT-1 STATE

FIN-WAIT-2 STATE

Strictly speaking, this is an error and should receive a

"error: connection closing" response. An "ok" response would

be acceptable, too, as long as a second FIN is not emitted (the

first FIN may be retransmitted though).

CLOSE-WAIT STATE

Queue this request until all preceding SENDs have been

segmentized; then send a FIN segment, enter LAST-ACK state.

CLOSING STATE

LAST-ACK STATE

TIME-WAIT STATE

Eddy Expires February 7, 2021 [Page 64]

Internet-Draft TCP Specification August 2020

Respond with "error: connection closing".

Eddy Expires February 7, 2021 [Page 65]

Internet-Draft TCP Specification August 2020

ABORT Call

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

If the user should not have access to such a connection, return

"error: connection illegal for this process".

Otherwise return "error: connection does not exist".

LISTEN STATE

Any outstanding RECEIVEs should be returned with "error:

connection reset" responses. Delete TCB, enter CLOSED state,

and return.

SYN-SENT STATE

All queued SENDs and RECEIVEs should be given "connection

reset" notification, delete the TCB, enter CLOSED state, and

return.

SYN-RECEIVED STATE

ESTABLISHED STATE

FIN-WAIT-1 STATE

FIN-WAIT-2 STATE

CLOSE-WAIT STATE

Send a reset segment:

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

All queued SENDs and RECEIVEs should be given "connection

reset" notification; all segments queued for transmission

(except for the RST formed above) or retransmission should be

flushed, delete the TCB, enter CLOSED state, and return.

CLOSING STATE LAST-ACK STATE TIME-WAIT STATE

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

return.

Eddy Expires February 7, 2021 [Page 66]

Internet-Draft TCP Specification August 2020

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.

Eddy Expires February 7, 2021 [Page 67]

Internet-Draft TCP Specification August 2020

SEGMENT ARRIVES

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

all data in the incoming segment is discarded. An incoming

segment containing a RST is discarded. An incoming segment not

containing a RST causes a RST to be sent in response. The

acknowledgment and sequence field values are selected to make

the reset sequence acceptable to the TCP 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.

If the state is LISTEN then

first check for an RST

An incoming RST should be ignored. Return.

second check for an ACK

Any acknowledgment is bad if it arrives on a connection

still in the LISTEN state. An acceptable reset segment

should be formed for any arriving ACK-bearing segment. The

RST should be formatted as follows:

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

Return.

third check for a SYN

If the SYN bit is set, check the security. If the security/

compartment on the incoming segment does not exactly match

the security/compartment in the TCB then send a reset and

return.

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

Eddy Expires February 7, 2021 [Page 68]

Internet-Draft TCP Specification August 2020

Set RCV.NXT to SEG.SEQ+1, IRS is set to SEG.SEQ and any

other control or text should be queued for processing later.

ISS should be selected and a SYN segment sent of the form:

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

SND.NXT is set to ISS+1 and SND.UNA to ISS. The connection

state should be changed to SYN-RECEIVED. Note that any

other incoming control or data (combined with SYN) will be

processed in the SYN-RECEIVED state, but processing of SYN

and ACK should not be repeated. If the listen was not fully

specified (i.e., the remote socket was not fully specified),

then the unspecified fields should be filled in now.

fourth other text or control

Any other control or text-bearing segment (not containing

SYN) must have an ACK and thus would be discarded by the ACK

processing. An incoming RST segment could not be valid,

since it could not have been sent in response to anything

sent by this incarnation of the connection. So, if this

unlikely condition is reached, the correct behavior is to

drop the segment and return.

If the state is SYN-SENT then

first check the ACK bit

If the ACK bit is set

If SEG.ACK =< ISS, or SEG.ACK > SND.NXT, send a reset

(unless the RST bit is set, if so drop the segment and

return)

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

and discard the segment. Return.

If SND.UNA < SEG.ACK =< SND.NXT then the ACK is

acceptable. Some deployed TCP code has used the check

SEG.ACK == SND.NXT (using "==" rather than "=<", but this

is not appropriate when the stack is capable of sending

data on the SYN, because the 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

Eddy Expires February 7, 2021 [Page 69]

Internet-Draft TCP Specification August 2020

A potential blind reset attack is described in RFC 5961

[32], with the mitigation that a TCP implementation

SHOULD first check that the sequence number exactly

matches RCV.NXT prior to executing the action in the next

paragraph.

If the ACK was acceptable then signal the user "error:

connection reset", drop the segment, enter CLOSED state,

delete TCB, and return. Otherwise (no ACK) drop the

segment and return.

third check the security

If the security/compartment in the segment does not exactly

match the security/compartment in the TCB, send a reset

If there is an ACK

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

Otherwise

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

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

fourth check the SYN bit

This step should be reached only if the ACK is ok, or there

is no ACK, and it the segment did not contain a RST.

If the SYN bit is on and the security/compartment is

acceptable then, RCV.NXT is set to SEG.SEQ+1, IRS is set to

SEG.SEQ. SND.UNA should be advanced to equal SEG.ACK (if

there is an ACK), and any segments on the retransmission

queue 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

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

and send it. Data or controls that were queued for

transmission may be included. If there are other controls

or text in the segment then continue processing at the sixth

step below where the URG bit is checked, otherwise return.

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

Eddy Expires February 7, 2021 [Page 70]

Internet-Draft TCP Specification August 2020

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

and send it. Set the variables:

SND.WND <- SEG.WND

SND.WL1 <- SEG.SEQ

SND.WL2 <- SEG.ACK

If there are other controls or text in the segment, queue

them for processing after the ESTABLISHED state has been

reached, return.

Note that it is legal to send and receive application data

on SYN segments (this is the "text in the segment" mentioned

above. There has been significant misinformation and

misunderstanding of this topic historically. Some firewalls

and security devices consider this suspicious. However, the

capability was used in T/TCP [17] and is used in TCP Fast

Open (TFO) [41], so is important for implementations and

network devices to permit.

fifth, if neither of the SYN or RST bits is set then drop the

segment and return.

Otherwise,

first check sequence number

SYN-RECEIVED STATE

ESTABLISHED STATE

FIN-WAIT-1 STATE

FIN-WAIT-2 STATE

CLOSE-WAIT STATE

CLOSING STATE

LAST-ACK STATE

TIME-WAIT STATE

Segments are processed in sequence. Initial tests on

arrival are used to discard old duplicates, but further

processing is done in SEG.SEQ order. If a segment's

contents straddle the boundary between old and new, only the

new parts should be processed.

In general, the processing of received segments MUST be

implemented to aggregate ACK segments whenever possible

(MUST-58). For example, if the TCP endpoint is processing a

series of queued segments, it MUST process them all before

sending any ACK segments (MUST-59).

Eddy Expires February 7, 2021 [Page 71]

Internet-Draft TCP Specification August 2020

There are four cases for the acceptability test for an

incoming segment:

Segment Receive Test

Length Window

------- ------- -------------------------------------------

0 0 SEG.SEQ = RCV.NXT

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

>0 0 not acceptable

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

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

In implementing sequence number validation as described

here, please note Appendix A.2.

If the RCV.WND is zero, no segments will be acceptable, but

special allowance should be made to accept valid ACKs, URGs

and RSTs.

If an incoming segment is not acceptable, an acknowledgment

should be sent in reply (unless the RST bit is set, if so

drop the segment and return):

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

After sending the acknowledgment, drop the unacceptable

segment and return.

Note that for the TIME-WAIT state, there is an improved

algorithm described in [34] for handling incoming SYN

segments, that utilizes timestamps rather than relying on

the sequence number check described here. When the improved

algorithm is implemented, the logic above is not applicable

for incoming SYN segments with timestamp options, received

on a connection in the TIME-WAIT state.

In the following it is assumed that the segment is the

idealized segment that begins at RCV.NXT and does not exceed

the window. One could tailor actual segments to fit this

assumption by trimming off any portions that lie outside the

window (including SYN and FIN), and only processing further

if the segment then begins at RCV.NXT. Segments with higher

Eddy Expires February 7, 2021 [Page 72]

Internet-Draft TCP Specification August 2020

beginning sequence numbers SHOULD be held for later

processing (SHLD-31).

second check the RST bit,

RFC 5961 section 3 describes a potential blind reset attack

and optional mitigation approach that SHOULD be implemented.

For stacks implementing RFC 5961, the three checks below

apply, otherwise 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.

SYN-RECEIVED STATE

If the RST bit is set

If this connection was initiated with a passive OPEN

(i.e., came from the LISTEN state), then return this

connection to LISTEN state and return. The user need

not be informed. If this connection was initiated

with an active OPEN (i.e., came from SYN-SENT state)

then the connection was refused, signal the user

"connection refused". In either case, all segments on

the retransmission queue should be removed. And in

the active OPEN case, enter the CLOSED state and

delete the TCB, and return.

Eddy Expires February 7, 2021 [Page 73]

Internet-Draft TCP Specification August 2020

ESTABLISHED

FIN-WAIT-1

FIN-WAIT-2

CLOSE-WAIT

If the RST bit is set then, any outstanding RECEIVEs and

SEND should receive "reset" responses. All segment

queues should be flushed. Users should also receive an

unsolicited general "connection reset" signal. Enter the

CLOSED state, delete the TCB, and return.

CLOSING STATE

LAST-ACK STATE

TIME-WAIT

If the RST bit is set then, enter the CLOSED state,

delete the TCB, and return.

third check security

SYN-RECEIVED

If the security/compartment in the segment does not

exactly match the security/compartment in the TCB then

send a reset, and return.

ESTABLISHED

FIN-WAIT-1

FIN-WAIT-2

CLOSE-WAIT

CLOSING

LAST-ACK

TIME-WAIT

If the security/compartment in the segment does not

exactly match the security/compartment in the TCB then

send a reset, any outstanding RECEIVEs and SEND should

receive "reset" responses. All segment queues should be

flushed. Users should also receive an unsolicited

general "connection reset" signal. Enter the CLOSED

state, delete the TCB, and return.

Note this check is placed following the sequence check to

prevent a segment from an old connection between these port numbers

with a different security from causing an abort of the

current connection.

Eddy Expires February 7, 2021 [Page 74]

Internet-Draft TCP Specification August 2020

fourth, check the SYN bit,

SYN-RECEIVED

If the connection was initiated with a passive OPEN, then

return this connection to the LISTEN state and return.

Otherwise, handle per the directions for synchronized

states below.

ESTABLISHED STATE

FIN-WAIT STATE-1

FIN-WAIT STATE-2

CLOSE-WAIT STATE

CLOSING STATE

LAST-ACK STATE

TIME-WAIT STATE

If the SYN bit is set in these synchronized states, it

may be either a legitimate new connection attempt (e.g.

in the case of TIME-WAIT), an error where the connection

should be reset, or the result of an attack attempt, as

described in RFC 5961 [32]. For the TIME-WAIT state, new

connections can be accepted if the timestamp option is

used and meets expectations (per [34]). For all other

cases, RFC 5961 provides a mitigation that SHOULD be

implemented, though there are alternatives (see

Section 6). RFC 5961 recommends that in these

synchronized states, if the SYN bit is set, irrespective

of the sequence number, TCP endpoints MUST send a

"challenge ACK" to the remote peer:

<SEQ=SND.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.

Eddy Expires February 7, 2021 [Page 75]

Internet-Draft TCP Specification August 2020

If the SYN is not in the window this step would not be

reached and an ACK would have been sent in the first step

(sequence number check).

fifth check the ACK field,

if the ACK bit is off drop the segment and return

if the ACK bit is on

RFC 5961 section 5 describes a potential blind data

injection attack, and mitigation that implementations MAY

choose to include (MAY-12). TCP stacks that implement

RFC 5961 MUST add an input check that the ACK value is

acceptable only if it is in the range of ((SND.UNA -

MAX.SND.WND) =< SEG.ACK =< SND.NXT). All incoming

segments whose ACK value doesn't satisfy the above

condition MUST be discarded and an ACK sent back. The

new state variable MAX.SND.WND is defined as the largest

window that the local sender has ever received from its

peer (subject to window scaling) or may be hard-coded to

a maximum permissible window value. When the ACK value

is acceptable, the processing per-state below applies:

SYN-RECEIVED STATE

If SND.UNA < SEG.ACK =< SND.NXT then enter ESTABLISHED

state and continue processing with variables below set

to:

SND.WND <- SEG.WND

SND.WL1 <- SEG.SEQ

SND.WL2 <- SEG.ACK

If the segment acknowledgment is not acceptable,

form a reset segment,

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

and send it.

ESTABLISHED STATE

If SND.UNA < SEG.ACK =< SND.NXT then, set SND.UNA <-

SEG.ACK. Any segments on the retransmission queue

that are thereby entirely acknowledged are removed.

Users should receive positive acknowledgments for

buffers that have been SENT and fully acknowledged

Eddy Expires February 7, 2021 [Page 76]

Internet-Draft TCP Specification August 2020

(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

Eddy Expires February 7, 2021 [Page 77]

Internet-Draft TCP Specification August 2020

acknowledged, delete the TCB, enter the CLOSED state,

and return.

TIME-WAIT STATE

The only thing that can arrive in this state is a

retransmission of the remote FIN. Acknowledge it, and

restart the 2 MSL timeout.

sixth, check the URG bit,

ESTABLISHED STATE

FIN-WAIT-1 STATE

FIN-WAIT-2 STATE

If the URG bit is set, RCV.UP <- max(RCV.UP,SEG.UP), and

signal the user that the remote side has urgent data if

the urgent pointer (RCV.UP) is in advance of the data

consumed. If the user has already been signaled (or is

still in the "urgent mode") for this continuous sequence

of urgent data, do not signal the user again.

CLOSE-WAIT STATE

CLOSING STATE

LAST-ACK STATE

TIME-WAIT

This should not occur, since a FIN has been received from

the remote side. Ignore the URG.

seventh, process the segment text,

ESTABLISHED STATE

FIN-WAIT-1 STATE

FIN-WAIT-2 STATE

Once in the ESTABLISHED state, it is possible to deliver

segment text to user RECEIVE buffers. Text from segments

can be moved into buffers until either the buffer is full

or the segment is empty. If the segment empties and

carries 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.

Eddy Expires February 7, 2021 [Page 78]

Internet-Draft TCP Specification August 2020

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 3.7.

Send an acknowledgment of the form:

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

This acknowledgment should be piggybacked on a segment

being transmitted if possible without incurring undue

delay.

CLOSE-WAIT STATE

CLOSING STATE

LAST-ACK STATE

TIME-WAIT STATE

This should not occur, since a FIN has been received from

the remote side. Ignore the segment text.

eighth, check the FIN bit,

Do not process the FIN if the state is CLOSED, LISTEN or

SYN-SENT since the SEG.SEQ cannot be validated; drop the

segment and return.

If the FIN bit is set, signal the user "connection closing"

and return any pending RECEIVEs with same message, advance

RCV.NXT over the FIN, and send an acknowledgment for the

FIN. Note that FIN implies PUSH for any segment text not

yet delivered to the user.

SYN-RECEIVED STATE

ESTABLISHED STATE

Enter the CLOSE-WAIT state.

FIN-WAIT-1 STATE

Eddy Expires February 7, 2021 [Page 79]

Internet-Draft TCP Specification August 2020

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.

Eddy Expires February 7, 2021 [Page 80]

Internet-Draft TCP Specification August 2020

USER TIMEOUT

USER TIMEOUT

For any state if the user timeout expires, flush all queues,

signal the user "error: connection aborted due to user timeout"

in general and for any outstanding calls, delete the TCB, enter

the CLOSED state and return.

RETRANSMISSION TIMEOUT

For any state if the retransmission timeout expires on a

segment in the retransmission queue, send the segment at the

front of the retransmission queue again, reinitialize the

retransmission timer, and return.

TIME-WAIT TIMEOUT

If the time-wait timeout expires on a connection delete the

TCB, enter the CLOSED state and return.

Eddy Expires February 7, 2021 [Page 81]

Internet-Draft TCP Specification August 2020

3.10. 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 remote endpoint.

FIN

A control bit (finis) occupying one sequence number, which

indicates that the sender will send no more data or control

occupying sequence space.

fragment

A portion of a logical unit of data, in particular an

internet fragment is a portion of an internet datagram.

header

Control information at the beginning of a message, segment,

fragment, packet or block of data.

host

A computer. In particular a source or destination of

messages from the point of view of the communication network.

Identification

An Internet Protocol field. This identifying value assigned

by the sender aids in assembling the fragments of a datagram.

internet address

A network layer address.

internet datagram

The unit of data exchanged between an internet module and the

higher level protocol together with the internet header.

internet fragment

Eddy Expires February 7, 2021 [Page 82]

Internet-Draft TCP Specification August 2020

A portion of the data of an internet datagram with an

internet header.

IP

Internet Protocol. See [1] and [12].

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

Eddy Expires February 7, 2021 [Page 83]

Internet-Draft TCP Specification August 2020

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 of this range are considered duplicates and

discarded.

RST

A control bit (reset), occupying no sequence space,

indicating that the receiver should delete the connection

without further interaction. The receiver can determine,

based on the sequence number and acknowledgment fields of the

incoming segment, whether it should honor the reset command

or ignore it. In no case does receipt of a segment

containing RST give rise to a RST in response.

SEG.ACK

segment acknowledgment

SEG.LEN

Eddy Expires February 7, 2021 [Page 84]

Internet-Draft TCP Specification August 2020

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

Eddy Expires February 7, 2021 [Page 85]

Internet-Draft TCP Specification August 2020

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

[5] containing the Differentiated Services Code Point (DSCP)

value and the 2-bit ECN codepoint [8].

Type of Service

An Internet Protocol field that indicates the type of service

for this internet fragment.

URG

A control bit (urgent), occupying no sequence space, used to

indicate that the receiving user should be notified to do

urgent processing as long as there is data to be consumed

with sequence numbers less than the value indicated in the

urgent pointer.

Eddy Expires February 7, 2021 [Page 86]

Internet-Draft TCP Specification August 2020

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.

4. Changes from RFC 793

This document obsoletes RFC 793 as well as RFC 6093 and 6528, which

updated 793. In all cases, only the normative protocol specification

and requirements have been incorporated into this document, and some

informational text with background and rationale may not have been

carried in. The informational content of those documents is still

valuable in learning about and understanding TCP, and they are valid

Informational references, even though their normative content has

been incorporated into this document.

The main body of this document was adapted from RFC 793's Section 3,

titled "FUNCTIONAL SPECIFICATION", with an attempt to keep formatting

and layout as close as possible.

The collection of applicable RFC Errata that have been reported and

either accepted or held for an update to RFC 793 were incorporated

(Errata IDs: 573, 574, 700, 701, 1283, 1561, 1562, 1564, 1565, 1571,

1572, 2296, 2297, 2298, 2748, 2749, 2934, 3213, 3300, 3301, 6222).

Some errata were not applicable due to other changes (Errata IDs:

572, 575, 1569, 3305, 3602).

Changes to the specification of the Urgent Pointer described in RFC

1122 and 6093 were incorporated. See RFC 6093 for detailed

discussion of why these changes were necessary.

The discussion of the RTO from RFC 793 was updated to refer to RFC

6298. The RFC 1122 text on the RTO originally replaced the 793 text,

however, RFC 2988 should have updated 1122, and has subsequently been

obsoleted by 6298.

RFC 1122 contains a collection of other changes and clarifications to

RFC 793. The normative items impacting the protocol have been

incorporated here, though some historically useful implementation

advice and informative discussion from RFC 1122 is not included here.

RFC 1122 contains more than just TCP requirements, so this document

can't obsolete RFC 1122 entirely. It is only marked as "updating"

1122, however, it should be understood to effectively obsolete all of

the RFC 1122 material on TCP.

Eddy Expires February 7, 2021 [Page 87]

Internet-Draft TCP Specification August 2020

The more secure Initial Sequence Number generation algorithm from RFC

6528 was incorporated. See RFC 6528 for discussion of the attacks

that this mitigates, as well as advice on selecting PRF algorithms

and managing secret key data.

A note based on RFC 6429 was added to explicitly clarify that system

resource management concerns allow connection resources to be

reclaimed. RFC 6429 is obsoleted in the sense that this

clarification has been reflected in this update to the base TCP

specification now.

RFC EDITOR'S NOTE: the content below is for detailed change tracking

and planning, and not to be included with the final revision of the

document.

This document started as draft-eddy-rfc793bis-00, that was merely a

proposal and rough plan for updating RFC 793.

The -01 revision of this draft-eddy-rfc793bis incorporates the

content of RFC 793 Section 3 titled "FUNCTIONAL SPECIFICATION".

Other content from RFC 793 has not been incorporated. The -01

revision of this document makes some minor formatting changes to the

RFC 793 content in order to convert the content into XML2RFC format

and account for left-out parts of RFC 793. For instance, figure

numbering differs and some indentation is not exactly the same.

The -02 revision of draft-eddy-rfc793bis incorporates errata that

have been verified:

Errata ID 573: Reported by Bob Braden (note: This errata basically

is just a reminder that RFC 1122 updates 793. Some of the

associated changes are left pending to a separate revision that

incorporates 1122. Bob's mention of PUSH in 793 section 2.8 was

not applicable here because that section was not part of the

"functional specification". Also the 1122 text on the

retransmission timeout also has been updated by subsequent RFCs,

so the change here deviates from Bob's suggestion to apply the

1122 text.)

Errata ID 574: Reported by Yin Shuming

Errata ID 700: Reported by Yin Shuming

Errata ID 701: Reported by Yin Shuming

Errata ID 1283: Reported by Pei-chun Cheng

Errata ID 1561: Reported by Constantin Hagemeier

Errata ID 1562: Reported by Constantin Hagemeier

Errata ID 1564: Reported by Constantin Hagemeier

Errata ID 1565: Reported by Constantin Hagemeier

Errata ID 1571: Reported by Constantin Hagemeier

Errata ID 1572: Reported by Constantin Hagemeier

Eddy Expires February 7, 2021 [Page 88]

Internet-Draft TCP Specification August 2020

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, 1122, and 6691, as well as

option-handling requirements from RFC 1122.

The -00 revision of draft-ietf-tcpm-rfc793bis incorporates several

additional clarifications and updates to the section on segmentation,

many of which are based on feedback from Joe Touch improving from the

initial text on this in the previous revision.

The -01 revision incorporates the change to Reserved bits due to ECN,

as well as many other changes that come from RFC 1122.

The -02 revision has small formatting modifications in order to

address xml2rfc warnings about long lines. It was a quick update to

avoid document expiration. TCPM working group discussion in 2015

also indicated that that we should not try to add sections on

implementation advice or similar non-normative information.

Eddy Expires February 7, 2021 [Page 89]

Internet-Draft TCP Specification August 2020

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/compartment

Eddy Expires February 7, 2021 [Page 90]

Internet-Draft TCP Specification August 2020

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.

Eddy Expires February 7, 2021 [Page 91]

Internet-Draft TCP Specification August 2020

Some other suggested changes that will not be incorporated in this

793 update unless TCPM consensus changes with regard to scope are:

1. Tony Sabatini's suggestion for describing DO field

2. Per discussion with Joe Touch (TAPS list, 6/20/2015), the

description of the API could be revisited

Early in the process of updating RFC 793, Scott Brim mentioned that

this should include a PERPASS/privacy review. This may be something

for the chairs or AD to request during WGLC or IETF LC.

5. IANA Considerations

In the "Transmission Control Protocol (TCP) Header Flags" registry,

IANA is asked to 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 Name Reference Assignment Notes

Offset

--- ---- --------- ----------------

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

8 CWR (Congestion Window Reduced) [RFC3168]

9 ECE (ECN-Echo) [RFC3168]

10 Urgent Pointer field significant (URG) (this document)

11 Acknowledgment field significant (ACK) (this document)

12 Push Function (PSH) (this document)

13 Reset the connection (RST) (this document)

14 Synchronize sequence numbers (SYN) (this document)

15 No more data from sender (FIN) (this document)

Eddy Expires February 7, 2021 [Page 92]

Internet-Draft TCP Specification August 2020

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.

6. Security and Privacy Considerations

The TCP design includes only rudimentary security features that

improve the robustness and reliability of connections and application

data transfer, but there are no built-in cryptographic capabilities

to support any form of privacy, authentication, or other typical

security functions. Non-cryptographic enhancements (e.g. [32]) 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

[32]). Applications typically utilize lower-layer (e.g. IPsec) and

upper-layer (e.g. TLS) protocols to provide security and privacy for

TCP connections and application data carried in TCP. Methods based

on TCP options have been developed as well, to support some security

capabilities.

In order to fully protect TCP connections (including their control

flags) IPsec or the TCP Authentication Option (TCP-AO) [31] are the

only current effective methods. Other methods discussed in this

section may protect the payload, but either only a subset of the

fields (e.g., tcpcrypt [52]) or none at all (e.g., TLS). Other security

features that have been added to TCP (e.g., ISN generation, sequence

number checks) 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 [25]. 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" [52] 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.

Eddy Expires February 7, 2021 [Page 93]

Internet-Draft TCP Specification August 2020

The TCP Roadmap [42] 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 [33] 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) [21] that act as mitigations to these attacks.

Other attacks are focused on exhausting the resources of a TCP

server. Examples include SYN flooding [24] or wasting resources on

non-progressing connections [35]. Operating systems commonly

implement mitigations for these attacks. Some common defenses also

utilize proxies, stateful firewalls, and other technologies outside

of the end-host TCP implementation.

7. Acknowledgements

This document is largely a revision of RFC 793, which Jon Postel was

the editor of. Due to his excellent work, it was able to last for

three decades before we felt the need to revise it.

Andre Oppermann was a contributor and helped to edit the first

revision of this document.

We are thankful for the assistance of the IETF TCPM working group

chairs, over the course of work on this document:

Michael Scharf

Yoshifumi Nishida

Pasi Sarolahti

Michael Tuexen

During the discussions of this work on the TCPM mailing list and in

working group meetings, helpful comments, critiques, and reviews were

received from (listed alphabetically): David Borman, Mohamed

Boucadair, Bob Briscoe, Neal Cardwell, Yuchung Cheng, Martin Duke,

Ted Faber, Gorry Fairhurst, Fernando Gont, Rodney Grimes, Mike Kosek,

Kevin Lahey, Kevin Mason, Matt Mathis, Jonathan Morton, Tommy Pauly,

Tom Petch, Hagen Paul Pfeifer, Anthony Sabatini, Michael Scharf, Greg

Skinner, Joe Touch, Michael Tuexen, Reji Varghese, Tim Wicinski,

Eddy Expires February 7, 2021 [Page 94]

Internet-Draft TCP Specification August 2020

Lloyd Wood, and Alex Zimmermann. Joe Touch provided additional help

in clarifying the description of segment size parameters and PMTUD/

PLPMTUD recommendations.

This document includes content from errata that were reported by

(listed chronologically): Yin Shuming, Bob Braden, Morris M. Keesan,

Pei-chun Cheng, Constantin Hagemeier, Vishwas Manral, Mykyta

Yevstifeyev, EungJun Yi, Botong Huang, Charles Deng.

8. References

8.1. Normative References

[1] Postel, J., "Internet Protocol", STD 5, RFC 791,

DOI 10.17487/RFC0791, September 1981,

<https://www.rfc-editor.org/info/rfc791>.

[2] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191,

DOI 10.17487/RFC1191, November 1990,

<https://www.rfc-editor.org/info/rfc1191>.

[3] McCann, J., Deering, S., and J. Mogul, "Path MTU Discovery

for IP version 6", RFC 1981, DOI 10.17487/RFC1981, August

1996, <https://www.rfc-editor.org/info/rfc1981>.

[4] Bradner, S., "Key words for use in RFCs to Indicate

Requirement Levels", BCP 14, RFC 2119,

DOI 10.17487/RFC2119, March 1997,

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

[5] Nichols, K., Blake, S., Baker, F., and D. Black,

"Definition of the Differentiated Services Field (DS

Field) in the IPv4 and IPv6 Headers", RFC 2474,

DOI 10.17487/RFC2474, December 1998,

<https://www.rfc-editor.org/info/rfc2474>.

[6] Borman, D., Deering, S., and R. Hinden, "IPv6 Jumbograms",

RFC 2675, DOI 10.17487/RFC2675, August 1999,

<https://www.rfc-editor.org/info/rfc2675>.

[7] Lahey, K., "TCP Problems with Path MTU Discovery",

RFC 2923, DOI 10.17487/RFC2923, September 2000,

<https://www.rfc-editor.org/info/rfc2923>.

[8] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition

of Explicit Congestion Notification (ECN) to IP",

RFC 3168, DOI 10.17487/RFC3168, September 2001,

<https://www.rfc-editor.org/info/rfc3168>.

Eddy Expires February 7, 2021 [Page 95]

Internet-Draft TCP Specification August 2020

[9] Paxson, V., Allman, M., Chu, J., and M. Sargent,

"Computing TCP's Retransmission Timer", RFC 6298,

DOI 10.17487/RFC6298, June 2011,

<https://www.rfc-editor.org/info/rfc6298>.

[10] Gont, F., "Deprecation of ICMP Source Quench Messages",

RFC 6633, DOI 10.17487/RFC6633, May 2012,

<https://www.rfc-editor.org/info/rfc6633>.

[11] 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>.

[12] Deering, S. and R. Hinden, "Internet Protocol, Version 6

(IPv6) Specification", STD 86, RFC 8200,

DOI 10.17487/RFC8200, July 2017,

<https://www.rfc-editor.org/info/rfc8200>.

8.2. Informative References

[13] Postel, J., "Transmission Control Protocol", STD 7,

RFC 793, DOI 10.17487/RFC0793, September 1981,

<https://www.rfc-editor.org/info/rfc793>.

[14] Nagle, J., "Congestion Control in IP/TCP Internetworks",

RFC 896, DOI 10.17487/RFC0896, January 1984,

<https://www.rfc-editor.org/info/rfc896>.

[15] 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>.

[16] 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>.

[17] 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>.

[18] 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>.

Eddy Expires February 7, 2021 [Page 96]

Internet-Draft TCP Specification August 2020

[19] 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>.

[20] 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>.

[21] 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>.

[22] 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>.

[23] 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>.

[24] Eddy, W., "TCP SYN Flooding Attacks and Common

Mitigations", RFC 4987, DOI 10.17487/RFC4987, August 2007,

<https://www.rfc-editor.org/info/rfc4987>.

[25] Touch, J., "Defending TCP Against Spoofing Attacks",

RFC 4953, DOI 10.17487/RFC4953, July 2007,

<https://www.rfc-editor.org/info/rfc4953>.

[26] 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>.

[27] Gont, F., "TCP's Reaction to Soft Errors", RFC 5461,

DOI 10.17487/RFC5461, February 2009,

<https://www.rfc-editor.org/info/rfc5461>.

[28] 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>.

[29] 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>.

Eddy Expires February 7, 2021 [Page 97]

Internet-Draft TCP Specification August 2020

[30] 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>.

[31] 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>.

[32] 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>.

[33] 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>.

[34] 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>.

[35] 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>.

[36] 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>.

[37] Borman, D., "TCP Options and Maximum Segment Size (MSS)",

RFC 6691, DOI 10.17487/RFC6691, July 2012,

<https://www.rfc-editor.org/info/rfc6691>.

[38] Touch, J., "Updated Specification of the IPv4 ID Field",

RFC 6864, DOI 10.17487/RFC6864, February 2013,

<https://www.rfc-editor.org/info/rfc6864>.

[39] Touch, J., "Shared Use of Experimental TCP Options",

RFC 6994, DOI 10.17487/RFC6994, August 2013,

<https://www.rfc-editor.org/info/rfc6994>.

[40] 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>.

Eddy Expires February 7, 2021 [Page 98]

Internet-Draft TCP Specification August 2020

[41] 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>.

[42] 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>.

[43] 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>.

[44] 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>.

[45] 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>.

[46] 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>.

[47] 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>.

[48] IANA, "Transmission Control Protocol (TCP) Parameters,

https://www.iana.org/assignments/tcp-parameters/tcp-

parameters.xhtml", 2019.

[49] IANA, "Transmission Control Protocol (TCP) Header Flags,

https://www.iana.org/assignments/tcp-header-flags/tcp-

header-flags.xhtml", 2019.

[50] Gont, F., "Processing of IP Security/Compartment and

Precedence Information by TCP", draft-gont-tcpm-tcp-

seccomp-prec-00 (work in progress), March 2012.

Eddy Expires February 7, 2021 [Page 99]

Internet-Draft TCP Specification August 2020

[51] Gont, F. and D. Borman, "On the Validation of TCP Sequence

Numbers", draft-gont-tcpm-tcp-seq-validation-02 (work in

progress), March 2015.

[52] Bittau, A., Giffin, D., Handley, M., Mazieres, D., Slack,

Q., and E. Smith, "Cryptographic protection of TCP Streams

(tcpcrypt)", draft-ietf-tcpinc-tcpcrypt-09 (work in

progress), November 2017.

[53] Touch, J. and W. Eddy, "TCP Extended Data Offset Option",

draft-ietf-tcpm-tcp-edo-10 (work in progress), July 2018.

[54] Minshall, G., "A Proposed Modification to Nagle's

Algorithm", draft-minshall-nagle-01 (work in progress),

June 1999.

[55] Dalal, Y. and C. Sunshine, "Connection Management in

Transport Protocols", Computer Networks Vol. 2, No. 6, pp.

454-473, December 1978.

Appendix A. Other Implementation Notes

This section includes additional notes and references on TCP

implementation decisions that are currently not a part of the RFC

series or included within the TCP standard. These items can be

considered by implementers, but there was not yet a consensus to

include them in the standard.

A.1. IP Security Compartment and Precedence

The IPv4 specification [1] includes a precedence value in the (now

obsoleted) Type of Service field (TOS) field. It was modified in

[16], and then obsoleted by the definition of Differentiated Services

(DiffServ) [5]. 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 requires checking the IP security compartment and precedence

on incoming TCP segments for consistency within a connection, and

with application requests. Each of these aspects of IP have become

outdated, without specific updates to RFC 793. The issues with

precedence were fixed by [19], which is Standards Track, and so this

present TCP specification includes those changes. However, the state

of IP security options that may be used by MLS systems is not as

clean.

Resetting connections when incoming packets do not meet expected

security compartment or precedence expectations has been recognized

Eddy Expires February 7, 2021 [Page 100]

Internet-Draft TCP Specification August 2020

as a possible attack vector [50], 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) [43].

The obsoleted TOS processing rules in TCP assumed bidirectional (or

symmetric) precedence values used on a connection, but the DiffServ

architecture is asymmetric. Problems with the old TCP logic in this

regard were described in [19] 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, 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 [28]. 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.

Eddy Expires February 7, 2021 [Page 101]

Internet-Draft TCP Specification August 2020

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 [51], 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 [51].

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 [54] that

improves the situation for these applications.

This modification is implemented in some common operating systems,

and does not impact TCP interoperability. Additionally, many

applications simply disable Nagle, since this is generally supported

by a socket option. The TCP standard has not been updated to include

this Nagle modification, but implementers may find it beneficial to

consider.

A.4. Low Water Mark 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.

Eddy Expires February 7, 2021 [Page 102]

Internet-Draft TCP Specification August 2020

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.

| | | | |S| |

| | | | |H| |F

| | | | |O|M|o

| | |S| |U|U|o

| | |H| |L|S|t

| |M|O| |D|T|n

| |U|U|M| | |o

| |S|L|A|N|N|t

| |T|D|Y|O|O|t

FEATURE | ReqID | | | |T|T|e

-------------------------------------------------|--------|-|-|-|-|-|--

| | | | | | |

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| | | | |

Eddy Expires February 7, 2021 [Page 103]

Internet-Draft TCP Specification August 2020

Pointer indicates first non-urgent octet | MUST-62|x| | | | |

Arbitrary length urgent data sequence | MUST-31|x| | | | |

Inform ALP asynchronously of urgent data | MUST-32|x| | | | |1

ALP can learn if/how much urgent data Q'd | MUST-33|x| | | | |1

ALP employ the urgent mechanism | SHLD-13| | | |x| |

| | | | | | |

TCP Options | | | | | | |

Support the mandatory option set | MUST-4 |x| | | | |

Receive TCP option in any segment | MUST-5 |x| | | | |

Ignore unsupported options | MUST-6 |x| | | | |

Cope with illegal option length | MUST-7 |x| | | | |

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| | | | |

| | | | | | |

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| | | | |

Eddy Expires February 7, 2021 [Page 104]

Internet-Draft TCP Specification August 2020

Accept SYN from TIME-WAIT state | MAY-2 | | |x| | |

Use Timestamps to reduce TIME-WAIT | SHLD-4 | |x| | | |

| | | | | | |

Retransmissions | | | | | | |

Implement RFC 5681 | MUST-19|x| | | | |

Retransmit with same IP ident | MAY-4 | | |x| | |

Karn's algorithm | MUST-18|x| | | | |

| | | | | | |

Generating ACK's: | | | | | | |

Aggregate whenever possible | MUST-58|x| | | | |

Queue out-of-order segments | SHLD-31| |x| | | |

Process all Q'd before send ACK | MUST-59|x| | | | |

Send ACK for out-of-order segment | MAY-13 | | |x| | |

Delayed ACK's | SHLD-18| |x| | | |

Delay < 0.5 seconds | MUST-40|x| | | | |

Every 2nd full-sized segment ACK'd | SHLD-19|x| | | | |

Receiver SWS-Avoidance Algorithm | MUST-39|x| | | | |

| | | | | | |

Sending data | | | | | | |

Configurable TTL | MUST-49|x| | | | |

Sender SWS-Avoidance Algorithm | MUST-38|x| | | | |

Nagle algorithm | SHLD-7 | |x| | | |

Application can disable Nagle algorithm | MUST-17|x| | | | |

| | | | | | |

Connection Failures: | | | | | | |

Negative advice to IP on R1 retxs | MUST-20|x| | | | |

Close connection on R2 retxs | MUST-20|x| | | | |

ALP can set R2 | MUST-21|x| | | | |1

Inform ALP of R1<=retxs<R2 | SHLD-9 | |x| | | |1

Recommended value for R1 | SHLD-10| |x| | | |

Recommended value for R2 | SHLD-11| |x| | | |

Same mechanism for SYNs | MUST-22|x| | | | |

R2 at least 3 minutes for SYN | MUST-23|x| | | | |

| | | | | | |

Send Keep-alive Packets: | MAY-5 | | |x| | |

- Application can request | MUST-24|x| | | | |

- Default is "off" | MUST-25|x| | | | |

- Only send if idle for interval | MUST-26|x| | | | |

- Interval configurable | MUST-27|x| | | | |

- Default at least 2 hrs. | MUST-28|x| | | | |

- Tolerant of lost ACK's | MUST-29|x| | | | |

- Send with no data | SHLD-12| |x| | | |

- Configurable to send garbage octet | MAY-6 | | |x| | |

| | | | | | |

IP Options | | | | | | |

Ignore options TCP doesn't understand | MUST-50|x| | | | |

Time Stamp support | MAY-10 | | |x| | |

Record Route support | MAY-11 | | |x| | |

Eddy Expires February 7, 2021 [Page 105]

Internet-Draft TCP Specification August 2020

Source Route: | | | | | | |

ALP can specify | MUST-51|x| | | | |1

Overrides src rt in datagram | MUST-52|x| | | | |

Build return route from src rt | MUST-53|x| | | | |

Later src route overrides | SHLD-24| |x| | | |

| | | | | | |

Receiving ICMP Messages from IP | MUST-54|x| | | | |

Dest. Unreach (0,1,5) => inform ALP | SHLD-25| |x| | | |

Dest. Unreach (0,1,5) => abort conn | MUST-56| | | | |x|

Dest. Unreach (2-4) => abort conn | SHLD-26| |x| | | |

Source Quench => silent discard | MUST-55|x| | | | |

Time Exceeded => tell ALP, don't abort | MUST-56| | | | |x|

Param Problem => tell ALP, don't abort | MUST-56| | | | |x|

| | | | | | |

Address Validation | | | | | | |

Reject OPEN call to invalid IP address | MUST-46|x| | | | |

Reject SYN from invalid IP address | MUST-63|x| | | | |

Silently discard SYN to bcast/mcast addr | MUST-57|x| | | | |

| | | | | | |

TCP/ALP Interface Services | | | | | | |

Error Report mechanism | MUST-47|x| | | | |

ALP can disable Error Report Routine | SHLD-20| |x| | | |

ALP can specify DiffServ field for sending | MUST-48|x| | | | |

Passed unchanged to IP | SHLD-22| |x| | | |

ALP can change DiffServ field during connection| SHLD-21| |x| | | |

ALP generally changing DiffServ during conn. | SHLD-23| | | |x| |

Pass received DiffServ field up to ALP | MAY-9 | | |x| | |

FLUSH call | MAY-14 | | |x| | |

Optional local IP addr parm. in OPEN | MUST-43|x| | | | |

| | | | | | |

RFC 5961 Support: | | | | | | |

Implement data injection protection | MAY-12 | | |x| | |

| | | | | | |

Explicit Congestion Notification: | | | | | | |

Support ECN | SHLD-8 | |x| | | |

-------------------------------------------------|--------|-|-|-|-|-|-

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

Author's Address

Wesley M. Eddy (editor)

MTI Systems

US

Email: wes@mti-systems.com

Eddy Expires February 7, 2021 [Page 106]