Internet Engineering Task Force (IETF)
STD: 7
Request for Comments: 9293
Obsoletes: 793, 879, 2873, 6093, 6429, 6528, 6691
Updates: 1011, 1122, 5961
Category: Standards Track
ISSN: 2070-1721

W. Eddy, Ed.
MTI Systems

August 2022

Transmission Control Protocol (TCP)

Abstract

This document specifies the Transmission Control Protocol (TCP). TCP is an important transport-layer protocol in the Internet protocol stack, and it has continuously evolved over decades of use and growth of the Internet. Over this time, a number of changes have been made to TCP as it was specified in RFC 793, though these have only been documented in a piecemeal fashion. This document collects and brings those changes together with the protocol specification from RFC 793. This document obsoletes RFC 793, as well as RFCs 879, 2873, 6093, 6429, 6528, and 6691 that updated parts of RFC 793. It updates RFCs 1011 and 1122, and it should be considered as a replacement for the portions of those documents dealing with TCP requirements. It also updates RFC 5961 by adding a small clarification in reset handling while in the SYN-RECEIVED state. The TCP header control bits from RFC 793 have also been updated based on RFC 3168.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9293.

Table of Contents

1. Purpose and Scope

In 1981, RFC 793 [16] was released, documenting the Transmission Control Protocol (TCP) and replacing earlier published specifications for TCP.

Since then, TCP has been widely implemented, and it 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 [49]. Over time, a number of errata have been filed against RFC 793. There have also been deficiencies found and resolved in security, performance, and many other aspects. The number of enhancements has grown over time across many separate documents. These were never accumulated together into a comprehensive update to the base specification.

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

Some companion documents are referenced for important algorithms that are used by TCP (e.g., for congestion control) but have not been completely included in this document. This is a conscious choice, as this base specification can be used with multiple additional algorithms that are developed and incorporated separately. This document focuses on the common basis that all TCP implementations must support in order to interoperate. Since some additional TCP features have become quite complicated themselves (e.g., advanced loss recovery and congestion control), future companion documents may

attempt to similarly bring these together.

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

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

2. Introduction

RFC 793 contains a discussion of the TCP design goals and provides examples of its operation, including examples of connection establishment, connection termination, and packet retransmission to repair losses.

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

The "TCP Roadmap" [49] provides a more extensive guide to the RFCs that define TCP and describe various important algorithms. The TCP Roadmap contains sections on strongly encouraged enhancements that improve performance and other aspects of TCP beyond the basic operation specified in this document. As one example, implementing congestion control (e.g., [8]) is a TCP requirement, but it 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 [47], but these are not strictly required or discussed in this document. Multipath considerations for TCP are also specified separately in [59].

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

2.1. Requirements Language

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

Each use of RFC 2119 keywords in the document is individually labeled and referenced in Appendix B, which summarizes implementation

requirements.

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

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

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

2.2. Key TCP Concepts

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

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

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

TCP supports unicast delivery of data. There are anycast applications that can successfully use TCP without modifications, though there is some risk of instability due to changes of lower-layer forwarding behavior [46].

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

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

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

A more detailed description of TCP features compared to other transport protocols can be found in Section 3.1 of [52]. Further description of the motivations for developing TCP and its role in the Internet protocol stack can be found in Section 2 of [16] and earlier versions of the TCP specification.

3. Functional Specification

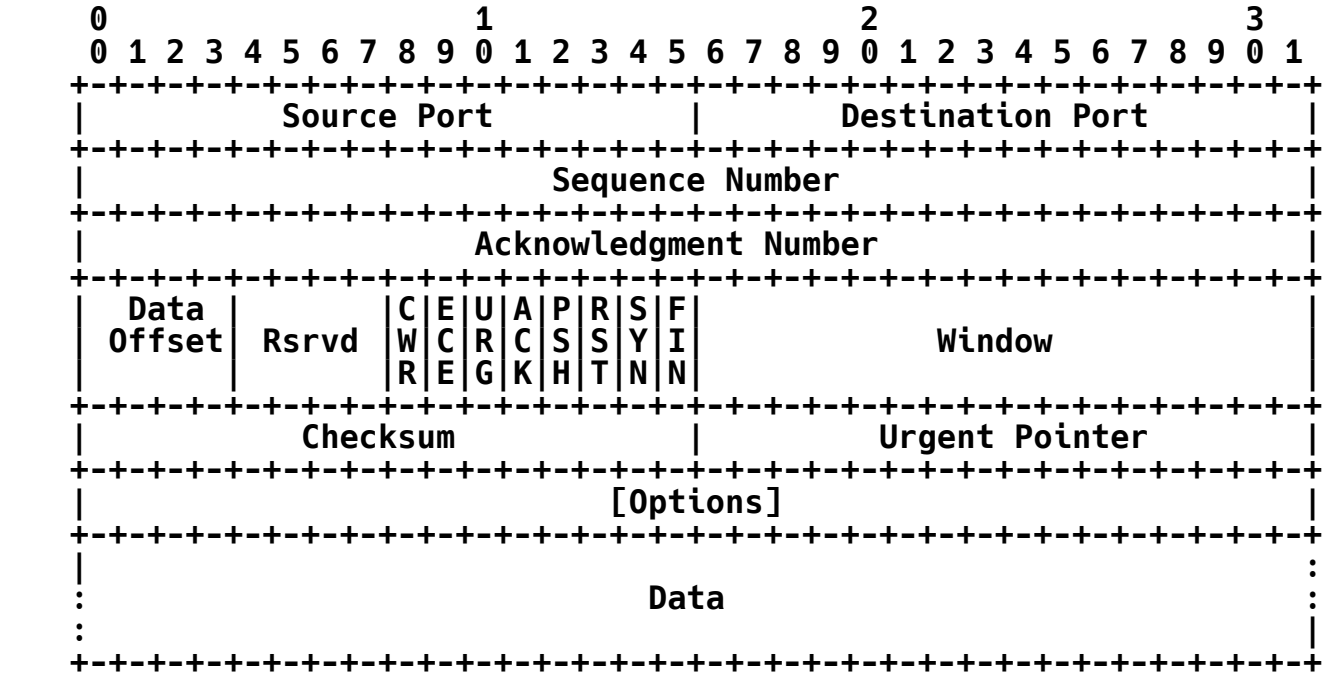
3.1. Header Format

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

fields had been a part of TCP.

This document describes TCP, which uses TCP headers.

A TCP header, followed by any user data in the segment, is formatted as follows, using the style from [66]:



Note that one tick mark represents one bit position.

Figure 1: TCP Header Format

where:

Source Port: 16 bits

  The source port number.

Destination Port: 16 bits

  The destination port number.

Sequence Number: 32 bits

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

Acknowledgment Number: 32 bits

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

Data Offset (DOffset): 4 bits

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

Reserved (Rsrvd): 4 bits

A set of control bits reserved for future use. Must be zero in generated segments and must be ignored in received segments if the corresponding future features are not implemented by the sending or receiving host.

Control bits: The control bits are also known as "flags". Assignment is managed by IANA from the "TCP Header Flags" registry [62]. The currently assigned control bits are CWR, ECE, URG, ACK, PSH, RST, SYN, and FIN.

CWR: 1 bit

Congestion Window Reduced (see [6]).

ECE: 1 bit

ECN-Echo (see [6]).

URG: 1 bit

Urgent pointer field is significant.

ACK: 1 bit

Acknowledgment field is significant.

PSH: 1 bit

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

RST: 1 bit

Reset the connection.

SYN: 1 bit

Synchronize sequence numbers.

FIN: 1 bit

No more data from sender.

Window: 16 bits

The number of data octets beginning with the one indicated in the acknowledgment field that the sender of this segment is willing to accept. The value is shifted when the window scaling extension is used [47].

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

Checksum: 16 bits

The checksum field is the 16-bit ones' complement of the ones' complement sum of all 16-bit words in the header and text. The checksum computation needs to ensure the 16-bit alignment of the data being summed. If a segment contains an odd number of header and text octets, alignment can be achieved by padding the last octet with zeros on its right to form a 16-bit word for checksum purposes. The pad is not transmitted as part of the segment. While computing the checksum, the checksum field itself is replaced with zeros.

The checksum also covers a pseudo-header (Figure 2) conceptually prefixed to the TCP header. The pseudo-header is 96 bits for IPv4 and 320 bits for IPv6. Including the pseudo-header in the checksum gives the TCP connection protection against misrouted segments. This information is carried in IP headers and is transferred across the TCP/network interface in the arguments or results of calls by the TCP implementation on the IP layer.

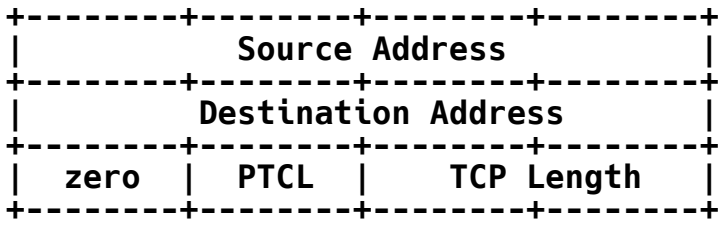


Figure 2: IPv4 Pseudo-header

Pseudo-header components for IPv4:

Source Address: the IPv4 source address in network byte order

Destination Address: the IPv4 destination address in network byte order

zero: bits set to zero

PTCL: the protocol number from the IP header

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

For IPv6, the pseudo-header is defined in Section 8.1 of RFC 8200 [13] and contains the IPv6 Source Address and Destination Address, an Upper-Layer Packet Length (a 32-bit value otherwise equivalent

to TCP Length in the IPv4 pseudo-header), three bytes of zero padding, and a Next Header value, which differs from the IPv6 header value if there are extension headers present between IPv6 and TCP.

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

Urgent Pointer: 16 bits

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

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

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

Case 1: A single octet of option-kind.

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

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

Note that the list of options may be shorter than the Data Offset field might imply. The content of the header beyond the End of Option List Option MUST be header padding of zeros (MUST-69).

The list of all currently defined options is managed by IANA [62], and each option is defined in other RFCs, as indicated there. That set includes experimental options that can be extended to support multiple concurrent usages [45].

A given TCP implementation can support any currently defined options, but the following options MUST be supported (MUST-4 -- note Maximum Segment Size Option support is also part of MUST-14 in Section 3.7.1):

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

Table 1: Mandatory Option Set

These options are specified in detail in Section 3.2.

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

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

Note: There is ongoing work to extend the space available for TCP Options, such as [65].

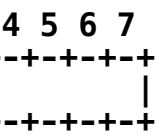
Data: variable length

User data carried by the TCP segment.

3.2. Specific Option Definitions

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

An End of Option List Option is formatted as follows:

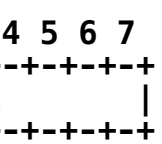


where:

Kind: 1 byte; Kind == 0.

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

A No-Operation Option is formatted as follows:

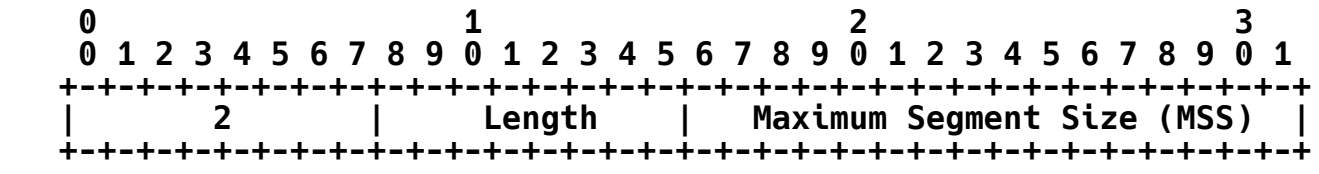


where:

Kind: 1 byte; Kind == 1.

This option code can be used between options, for example, to align the beginning of a subsequent option on a word boundary. There is no guarantee that senders will use this option, so receivers MUST be prepared to process options even if they do not begin on a word boundary (MUST-64).

A Maximum Segment Size Option is formatted as follows:



where:

Kind: 1 byte; Kind == 2.

If this option is present, then it communicates the maximum receive segment size at the TCP endpoint that sends this segment. This value is limited by the IP reassembly limit. This field may be sent in the initial connection request (i.e., in segments with the SYN control bit set) and MUST NOT be sent in other segments (MUST-65). If this option is not used, any segment size is allowed. A more complete description of this option is provided in Section 3.7.1.

Length: 1 byte; Length == 4.

Length of the option in bytes.

Maximum Segment Size (MSS): 2 bytes.

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

3.2.1. Other Common Options

Additional RFCs define some other commonly used options that are recommended to implement for high performance but are not necessary for basic TCP interoperability. These are the TCP Selective Acknowledgment (SACK) Option [22] [26], TCP Timestamp (TS) Option [47], and TCP Window Scale (WS) Option [47].

3.2.2. Experimental TCP Options

Experimental TCP Option values are defined in [30], and [45] describes the current recommended usage for these experimental values.

3.3. TCP Terminology Overview

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

is a glossary of terms in Section 4.

3.3.1. Key Connection State Variables

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

Variable	Description
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

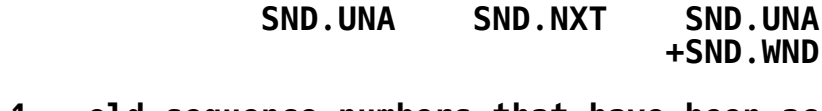
Table 2: Send Sequence Variables

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

Table 3: Receive Sequence Variables

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

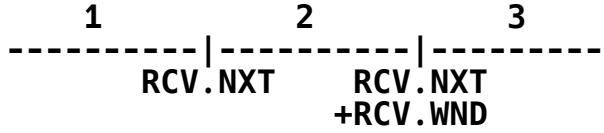
```
                1         2          3          4
           ----------|----------|----------|----------
```



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

Figure 3: Send Sequence Space

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



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

Figure 4: Receive Sequence Space

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

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

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

Table 4: Current Segment Variables

3.3.2. State Machine Overview

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

LISTEN - represents waiting for a connection request from any remote

TCP peer and port.

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

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

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

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

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

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

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

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

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

CLOSED - represents no connection state at all.

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

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

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

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

state names are abbreviated or hyphenated differently in the diagram from how they appear elsewhere in the document.

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

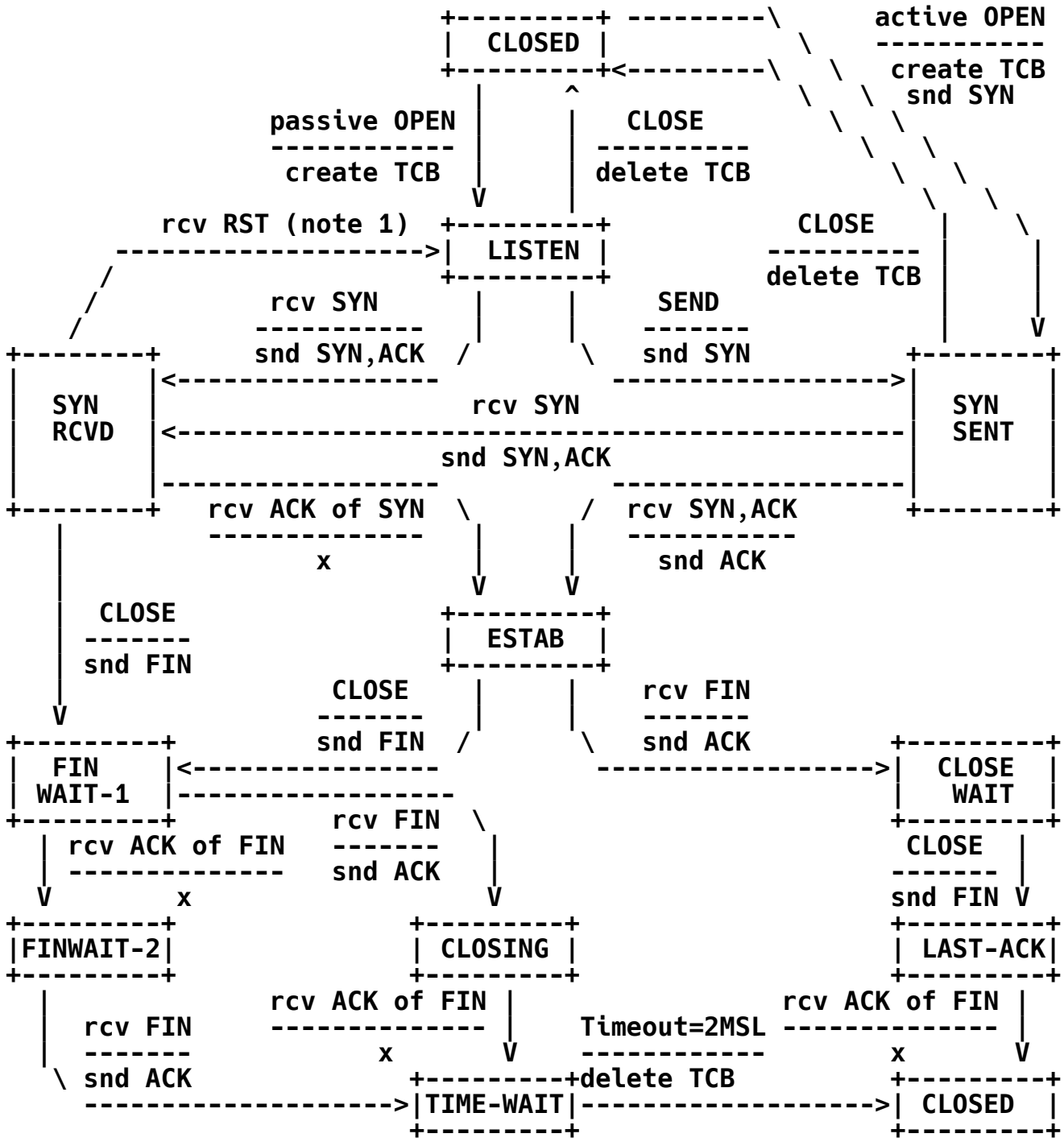


Figure 5: TCP Connection State Diagram

The following notes apply to Figure 5:

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

RST is conditional on having reached SYN-RECEIVED after a passive OPEN.

Note 2: The figure omits a transition from FIN-WAIT-1 to TIME-WAIT if a FIN is received and the local FIN is also acknowledged.

Note 3: A RST can be sent from any state with a corresponding transition to TIME-WAIT (see [70] for rationale). These transitions are not explicitly shown; otherwise, the diagram would become very difficult to read. Similarly, receipt of a RST from any state results in a transition to LISTEN or CLOSED, though this is also omitted from the diagram for legibility.

3.4. Sequence Numbers

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

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

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

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

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

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

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

SND.UNA = oldest unacknowledged sequence number

SND.NXT = next sequence number to be sent

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

SEG.SEQ = first sequence number of a segment

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

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

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

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 than or equal to the acknowledgment value in the incoming segment.

When data is received, the following comparisons are needed:

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

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

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

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

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

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

or

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

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

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

Segment Length	Receive Window	Test
0	0	SEG.SEQ = RCV.NXT

0	>0	RCV.NXT =< SEG.SEQ < RCV.NXT+RCV.WND
>0	0	not acceptable
>0	>0	RCV.NXT =< SEG.SEQ < RCV.NXT+RCV.WND or RCV.NXT =< SEG.SEQ+SEG.LEN-1 < RCV.NXT+RCV.WND

Table 5: Segment Acceptability Tests

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

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

3.4.1. Initial Sequence Number Selection

A connection is defined by a pair of sockets. Connections can be reused. New instances of a connection will be referred to as incarnations of the connection. The problem that arises from this is -- "how does the TCP implementation identify duplicate segments from previous incarnations of the connection?" This problem becomes apparent if the connection is being opened and closed in quick succession, or if the connection breaks with loss of memory and is then reestablished. To support this, the TIME-WAIT state limits the rate of connection reuse, while the initial sequence number selection described below further protects against ambiguity about which incarnation of a connection an incoming packet corresponds to.

To avoid confusion, we must prevent segments from one incarnation of a connection from being used while the same sequence numbers may still be present in the network from an earlier incarnation. We want

to assure this even if a TCP endpoint loses all knowledge of the sequence numbers it has been using. When new connections are created, an initial sequence number (ISN) generator is employed that selects a new 32-bit ISN. There are security issues that result if an off-path attacker is able to predict or guess ISN values [42].

TCP initial sequence numbers are generated from a number sequence that monotonically increases until it wraps, known loosely as a "clock". This clock is a 32-bit counter that typically increments at least once every roughly 4 microseconds, although it is neither assumed to be realtime nor precise, and need not persist across reboots. The clock component is intended to ensure that with a Maximum Segment Lifetime (MSL), generated ISNs will be unique since it cycles approximately every 4.55 hours, which is much longer than the MSL. Please note that for modern networks that support high data rates where the connection might start and quickly advance sequence numbers to overlap within the MSL, it is recommended to implement the Timestamp Option as mentioned later in Section 3.4.3.

A TCP implementation MUST use the above type of "clock" for clock-driven selection of initial sequence numbers (MUST-8), and SHOULD generate its initial sequence numbers with the expression:

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 [42].

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

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

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

1) A --> B SYN my sequence number is X

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

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

A 3WHS is necessary because sequence numbers are not tied to a global clock in the network, and TCP implementations may have different mechanisms for picking the ISNs. The receiver of the first SYN has no way of knowing whether the segment was an old one or not, unless it remembers the last sequence number used on the connection (which is not always possible), and so it must ask the sender to verify this SYN. The three-way handshake and the advantages of a clock-driven scheme for ISN selection are discussed in [69].

3.4.2. Knowing When to Keep Quiet

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

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

3.4.3. The TCP Quiet Time Concept

Hosts that for any reason lose knowledge of the last sequence numbers transmitted on each active (i.e., not closed) connection shall delay emitting any TCP segments for at least the agreed MSL in the internet system that the host is a part of. In the paragraphs below, an explanation for this specification is given. TCP implementers may violate the "quiet time" restriction, but only at the risk of causing some old data to be accepted as new or new data rejected as old duplicated data by some receivers in the internet system.

TCP endpoints consume sequence number space each time a segment is formed and entered into the network output queue at a source host. The duplicate detection and sequencing algorithm in TCP 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 reusing a sequence number before its first use has been acknowledged. This alone does not guarantee that old duplicate data is drained from the net, so the sequence space has been made large to reduce the probability that a wandering duplicate will cause trouble upon arrival. At 2 megabits/sec., it takes 4.5 hours to use up 2^32 octets of sequence space. Since the maximum segment lifetime in the net is not likely to exceed a few tens of seconds, this is deemed ample protection for foreseeable nets, even if data rates escalate to 10s of megabits/sec. At 100 megabits/sec., the cycle time is 5.4 minutes, which may be a little short but still within reason. Much higher data rates are possible today, with implications described in the final paragraph of this subsection.

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

Even hosts that can remember the time of day and use 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 and are willing to risk possible confusion of old and new packets at a given destination may choose not to wait for the "quiet time". Implementers 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, and the numbers occupied by a segment are "busy" or "in use" until MSL seconds have passed. Upon rebooting, a block of space-time is occupied by the octets and SYN or FIN flags of any potentially still in-flight segments. If a new connection is started too soon and uses any of the sequence numbers in the space-time footprint of those potentially still in-flight segments of the previous connection incarnation, there is a potential sequence number overlap area that could cause confusion at the receiver.

High-performance cases will have shorter cycle times than those in the megabits per second that the base TCP design described above considers. At 1 Gbps, the cycle time is 34 seconds, only 3 seconds at 10 Gbps, and around a third of a second at 100 Gbps. In these higher-performance cases, TCP Timestamp Options and Protection Against Wrapped Sequences (PAWS) [47] provide the needed capability to detect and discard old duplicates.

3.5. Establishing a Connection

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

Several examples of connection initiation follow. Although these examples do not show connection synchronization using data-carrying segments, this is perfectly legitimate, so long as the receiving TCP endpoint doesn't deliver the data to the user until it is clear the

data is valid (e.g., the data is buffered at the receiver until the connection reaches the ESTABLISHED state, given that the three-way handshake reduces the possibility of false connections). It is a trade-off between memory and messages to provide information for this checking.

The simplest 3WHS is shown in Figure 6. The figures should be interpreted in the following way. Each line is numbered for reference purposes. Right arrows (-->) indicate departure of a TCP segment from TCP Peer A to TCP Peer B or arrival of a segment at B from A. Left arrows (<--) indicate the reverse. Ellipses (...) indicate a segment that is still in the network (delayed). Comments appear in parentheses. TCP connection states represent the state AFTER the departure or arrival of the segment (whose contents are shown in the center of each line). Segment contents are shown in abbreviated form, with sequence number, control flags, and ACK field. Other fields such as window, addresses, lengths, and text have been left out in the interest of clarity.

```
    TCP Peer A                                                TCP Peer B

1.  CLOSED                                                    LISTEN

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

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

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

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

Figure 6: Basic Three-Way Handshake for Connection Synchronization

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

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

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

```
    TCP Peer A                                            TCP Peer B

1.  CLOSED                                                CLOSED

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

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

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

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

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

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

Figure 7: Simultaneous Connection Synchronization

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

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

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

```
    TCP Peer A                                                TCP Peer B

1.  CLOSED                                                    LISTEN

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

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

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

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

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

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

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

Figure 8: Recovery from Old Duplicate SYN

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

at line 6, the synchronization proceeds normally. If the SYN at line 6 had arrived before the RST, a more complex exchange might have occurred with RSTs sent in both directions.

3.5.1. Half-Open Connections and Other Anomalies

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

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

Assume that two user processes A and B are communicating with one another when a failure or reboot occurs causing loss of memory to A's TCP implementation. Depending on the operating system supporting A's TCP implementation, it is likely that some error recovery mechanism exists. When the TCP endpoint is up again, A is likely to start again from the beginning or from a recovery point. As a result, A will probably try to OPEN the connection again or try to SEND on the connection it believes open. In the latter case, it receives the error message "connection not open" from the local (A's) TCP implementation. In an attempt to establish the connection, A's TCP implementation will send a segment containing SYN. This scenario leads to the example shown in Figure 9. After TCP Peer A reboots, the user attempts to reopen the connection. TCP Peer B, in the meantime, thinks the connection is open.

```
      TCP Peer A                                          TCP Peer B

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

  2.  CLOSED                                               ESTABLISHED

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

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

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

  6.  SYN-SENT                                             CLOSED

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

Figure 9: Half-Open Connection Discovery

When the SYN arrives at line 3, TCP Peer B, being in a synchronized state, and the incoming segment outside the window, responds with an acknowledgment indicating what sequence it next expects to hear (ACK

100). TCP Peer A sees that this segment does not acknowledge anything it sent and, being unsynchronized, sends a reset (RST) because it has detected a half-open connection. TCP Peer B aborts at line 5. TCP Peer A will continue to try to establish the connection; the problem is now reduced to the basic three-way handshake of Figure 6.

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

```
      TCP Peer A                                          TCP Peer B

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

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

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

Figure 10: Active Side Causes Half-Open Connection Discovery

In Figure 11, two TCP Peers A and B with passive connections waiting for SYN are depicted. An old duplicate arriving at TCP Peer B (line 2) stirs B into action. A SYN-ACK is returned (line 3) and causes TCP A to generate a RST (the ACK in line 3 is not acceptable). TCP Peer B accepts the reset and returns to its passive LISTEN state.

```
    TCP Peer A                                          TCP Peer B

1.  LISTEN                                              LISTEN

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

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

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

5.  LISTEN                                              LISTEN
```

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

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

3.5.2. Reset Generation

A TCP user or application can issue a reset on a connection at any time, though reset events are also generated by the protocol itself when various error conditions occur, as described below. The side of a connection issuing a reset should enter the TIME-WAIT state, as this generally helps to reduce the load on busy servers for reasons described in [70].

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

There are three groups of states:

1. If the connection does not exist (CLOSED), then a reset is sent in response to any incoming segment except another reset. A SYN segment that does not match an existing connection is rejected by this means.

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

2. If the connection is in any non-synchronized state (LISTEN, SYN-SENT, SYN-RECEIVED), and the incoming segment acknowledges something not yet sent (the segment carries an unacceptable ACK), or if an incoming segment has a security level or compartment (Appendix A.1) that does not exactly match the level and compartment requested for the connection, a reset is sent.

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

3. If the connection is in a synchronized state (ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT), any unacceptable segment (out-of-window sequence number or unacceptable acknowledgment number) must be responded to with an empty acknowledgment segment (without any user data) containing the current send sequence number and an acknowledgment indicating the next sequence number expected to be received, and the connection remains in the same state.

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

3.5.3. Reset Processing

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

The receiver of a RST first validates it, then changes state. If the receiver was in the LISTEN state, it ignores it. If the receiver was

in SYN-RECEIVED state and had previously been in the LISTEN state, then the receiver returns to the LISTEN state; otherwise, the receiver aborts the connection and goes to the CLOSED state. If the receiver was in any other state, it aborts the connection and advises the user and goes to the CLOSED state.

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

3.6. Closing a Connection

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

There are essentially three cases:

1) The user initiates by telling the TCP implementation to CLOSE the connection (TCP Peer A in Figure 12).

2) The remote TCP endpoint initiates by sending a FIN control signal (TCP Peer B in Figure 12).

3) Both users CLOSE simultaneously (Figure 13).

Case 1: Local user initiates the close

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

Case 2: TCP endpoint receives a FIN from the network

If an unsolicited FIN arrives from the network, the receiving TCP

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

Case 3: Both users close simultaneously

A simultaneous CLOSE by users at both ends of a connection causes FIN segments to be exchanged (Figure 13). When all segments preceding the FINs have been processed and acknowledged, each TCP peer can ACK the FIN it has received. Both will, upon receiving these ACKs, delete the connection.

```
    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 12: 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 13: Simultaneous Close Sequence

A TCP connection may terminate in two ways: (1) the normal TCP close sequence using a FIN handshake (Figure 12), and (2) an "abort" in

which one or more RST segments are sent and the connection state is immediately discarded. If the local TCP connection is closed by the remote side due to a FIN or RST received from the remote side, then the local application MUST be informed whether it closed normally or was aborted (MUST-12).

3.6.1. Half-Closed Connections

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

A host MAY implement a "half-duplex" TCP close sequence, so that an application that has called CLOSE cannot continue to read data from the connection (MAY-1). If such a host issues a CLOSE call while received data is still pending in the TCP connection, or if new data is received after CLOSE is called, its TCP implementation SHOULD send a RST to show that data was lost (SHLD-3). See [23], Section 2.17 for discussion.

When a connection is closed actively, it MUST linger in the 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 [40] in order to support higher connection establishment rates. This algorithm for reducing TIME-WAIT is a Best Current Practice that SHOULD be implemented since Timestamp Options are commonly used, and using them to reduce TIME-WAIT provides benefits for busy Internet servers (SHLD-4).

3.7. Segmentation

The term "segmentation" refers to the activity TCP performs when ingesting a stream of bytes from a sending application and packetizing that stream of bytes into TCP segments. Individual TCP segments often do not correspond one-for-one to individual send (or socket write) calls from the application. Applications may perform writes at the granularity of messages in the upper-layer protocol, but TCP guarantees no correlation between the boundaries of TCP segments sent and received and the boundaries of the read or write buffers of user application data. In some specific protocols, such as Remote Direct Memory Access (RDMA) using Direct Data Placement (DDP) and Marker PDU Aligned Framing (MPA) [34], there are

performance optimizations possible when the relation between TCP segments and application data units can be controlled, and MPA includes a specific mechanism for detecting and verifying this relationship between TCP segments and application message data structures, but this is specific to applications like RDMA. In general, multiple goals influence the sizing of TCP segments created by a TCP implementation.

Goals driving the sending of larger segments include:

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

* Increasing processing efficiency and potential performance by enabling a smaller number of interrupts and inter-layer interactions.

* Limiting the overhead of TCP headers.

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

Goals driving the sending of smaller segments include:

* Avoiding sending a TCP segment that would result in an IP datagram larger than the smallest MTU along an IP network path because this results in either packet loss or packet fragmentation. Making matters worse, some firewalls or middleboxes may drop fragmented packets or ICMP messages related to fragmentation.

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

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

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

3.7.1. Maximum Segment Size Option

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

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

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

The maximum size of a segment that a 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 [19]) and the largest transmission size permitted by the IP layer (EMTU_S [19]):

Eff.snd.MSS = min(SendMSS+20, MMS_S) - TCPhdrsize - IPoptionsize

where:

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

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

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

* IPoptionsize is the size of any IPv4 options or IPv6 extension headers associated with a TCP connection. Note that some options or extension headers might not be included on all packets, but that for each segment sent, the sender should adjust the data length accordingly, within the Eff.snd.MSS.

The MSS value to be sent in an MSS Option should be equal to the effective MTU minus the fixed IP and TCP headers. By ignoring both IP and TCP Options when calculating the value for the MSS Option, if there are any IP or TCP Options to be sent in a packet, then the sender must decrease the size of the TCP data accordingly. RFC 6691 [43] 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 [19].

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

6691, Section 3.1.

3.7.2. Path MTU Discovery

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

PMTUD for IPv4 [2] or IPv6 [14] is implemented in conjunction between TCP, IP, and ICMP. 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 [27]. PLPMTUD [31] is a Standards Track improvement to PMTUD that relaxes the requirement for ICMP support across a path, and improves performance in cases where ICMP is not consistently conveyed, but still tries to avoid source fragmentation. The mechanisms in all four of these RFCs are recommended to be included in TCP implementations.

The TCP MSS Option specifies an upper bound for the size of packets that can be received (see [43]). Hence, setting the value in the MSS Option too small can impact the ability for PMTUD or PLPMTUD to find a larger path MTU. RFC 1191 discusses this implication of many older TCP implementations setting the TCP MSS to 536 (corresponding to the IPv4 576 byte default MTU) for non-local destinations, rather than deriving it from the MTUs of connected interfaces as recommended.

3.7.3. Interfaces with Variable MTU Values

The effective MTU can sometimes vary, as when used with variable compression, e.g., RObust Header Compression (ROHC) [37]. It is tempting for a TCP implementation to advertise the largest possible MSS, to support the most efficient use of compressed payloads. Unfortunately, some compression schemes occasionally need to transmit full headers (and thus smaller payloads) to resynchronize state at their endpoint compressors/decompressors. If the largest MTU is used to calculate the value to advertise in the MSS Option, TCP retransmission may interfere with compressor resynchronization.

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

3.7.4. Nagle Algorithm

The "Nagle algorithm" was described in RFC 896 [17] and was recommended in RFC 1122 [19] for mitigation of an early problem of too many small packets being generated. It has been implemented in most current TCP code bases, sometimes with minor variations (see Appendix A.3).

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

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

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

3.7.5. IPv6 Jumbograms

In order to support TCP over IPv6 Jumbograms, implementations need to be able to send TCP segments larger than the 64-KB limit that the MSS Option can convey. RFC 2675 [24] defines that an MSS value of 65,535 bytes is to be treated as infinity, and Path MTU Discovery [14] is used to determine the actual MSS.

The Jumbo Payload Option need not be implemented or understood by IPv6 nodes that do not support attachment to links with an MTU greater than 65,575 [24], and the present IPv6 Node Requirements does not include support for Jumbograms [55].

3.8. Data Communication

Once the connection is established, data is communicated by the exchange of segments. Because segments may be lost due to errors (checksum test failure) or network congestion, TCP uses retransmission to ensure delivery of every segment. Duplicate segments may arrive due to network or TCP retransmission. As discussed in the section on sequence numbers (Section 3.4), the TCP implementation performs certain tests on the sequence and acknowledgment numbers in the segments to verify their acceptability.

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

When the sender creates a segment and transmits it, the sender advances SND.NXT. When the receiver accepts a segment, it advances

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

The CLOSE user call implies a push function (see Section 3.9.1), as does the FIN control flag in an incoming segment.

3.8.1. Retransmission Timeout

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

The RTO MUST be computed according to the algorithm in [10], including Karn's algorithm for taking RTT samples (MUST-18).

RFC 793 contains an early example procedure for computing the RTO, based on work mentioned in IEN 177 [71]. This was then replaced by the algorithm described in RFC 1122, which was 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 [44]. TCP implementations should not rely on or typically interact with this IPv4 header field in any way. It is not a reasonable way to indicate duplicate sent segments nor to identify duplicate received segments.

3.8.2. TCP Congestion Control

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

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

A TCP endpoint MUST implement the basic congestion control algorithms slow start, congestion avoidance, and exponential backoff of RTO to avoid creating congestion collapse conditions (MUST-19). RFC 5681

and RFC 6298 describe the basic algorithms on the IETF Standards Track that are broadly applicable. Multiple other suitable algorithms exist and have been widely used. Many TCP implementations support a set of alternative algorithms that can be configured for use on the endpoint. An endpoint MAY implement such alternative algorithms provided that the algorithms are conformant with the TCP specifications from the IETF Standards Track as described in RFC 2914, RFC 5033 [7], and RFC 8961 [15] (MAY-18).

Explicit Congestion Notification (ECN) was defined in RFC 3168 and is an IETF Standards Track enhancement that has many benefits [51].

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

3.8.3. TCP Connection Failures

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

(a) There are two thresholds R1 and R2 measuring the amount of retransmission that has occurred for the same segment. R1 and R2 might be measured in time units or as a count of retransmissions (with the current RTO and corresponding backoffs as a conversion factor, if needed).

(b) When the number of transmissions of the same segment reaches or exceeds threshold R1, pass negative advice (see Section 3.3.1.4 of [19]) 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 the "Asynchronous Reports" section (Section 3.9.1.8)), when R1 is reached and before R2 (SHLD-9). This will allow a remote login application program to inform the user, for example.

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

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

notification of the application layer.

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

3.8.4. TCP Keep-Alives

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

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

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

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

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

3.8.5. The Communication of Urgent Information

As a result of implementation differences and middlebox interactions, new applications SHOULD NOT employ the TCP urgent mechanism (SHLD-13). However, TCP implementations MUST still include support for the urgent mechanism (MUST-30). Information on how some TCP implementations interpret the urgent pointer can be found in RFC 6093 [39].

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, then the TCP implementation must tell the user to go into "urgent mode"; when the receive sequence number catches up to the urgent pointer, the TCP implementation must tell user to go into "normal mode". If the urgent pointer is updated while the user is in "urgent mode", the update will be invisible to the user.

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

To send an urgent indication, the user must also send at least one data octet. If the sending user also indicates a push, timely delivery of the urgent information to the destination process is enhanced. Note that because changes in the urgent pointer correspond to data being written by a sending application, the urgent pointer cannot "recede" in the sequence space, but a TCP receiver should be robust to invalid urgent pointer values.

A TCP implementation MUST support a sequence of urgent data of any length (MUST-31) [19].

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

A TCP implementation MUST (MUST-32) inform the application layer asynchronously whenever it receives an urgent pointer and there was previously no pending urgent data, or whenever the urgent pointer advances in the data stream. The TCP implementation MUST (MUST-33) provide a way for the application to learn how much urgent data remains to be read from the connection, or at least to determine whether more urgent data remains to be read [19].

3.8.6. Managing the Window

The window sent in each segment indicates the range of sequence numbers the sender of the window (the data receiver) is currently prepared to accept. There is an assumption that this is related to the data buffer space currently 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 an acknowledgment number equal to or greater than the highest previously received).

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

The mechanisms provided allow a TCP endpoint to advertise a large window and to subsequently advertise a much smaller window without having accepted that much data. This so-called "shrinking the window" is strongly discouraged. The robustness principle [19] dictates that TCP peers will not shrink the window themselves, but will be prepared for such behavior on the part of other TCP peers.

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

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

3.8.6.1. Zero-Window Probing

The sending TCP peer must regularly transmit at least one octet of new data (if available), or retransmit to the receiving TCP peer even if the send window is zero, in order to "probe" the window. This retransmission is essential to guarantee that when either TCP peer has a zero window the reopening 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 [19]. The behavior is subject to the implementation's resource management concerns, as noted in [41].

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

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

3.8.6.2. Silly Window Syndrome Avoidance

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

3.8.6.2.1. Sender's Algorithm -- When to Send Data

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

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

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

The "usable window" is:

```
U = 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:

(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 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 3.8.6.1).

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

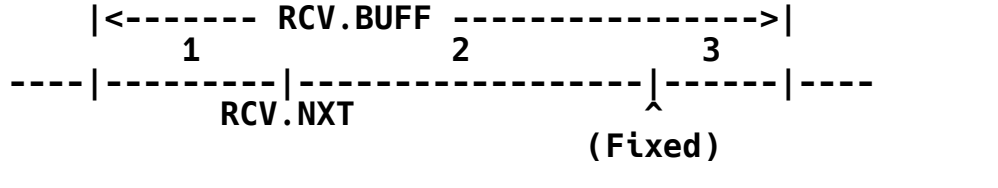
A TCP implementation MUST include a SWS avoidance algorithm in the receiver (MUST-39).

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

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



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

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

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

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

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

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

3.8.6.3. Delayed Acknowledgments -- When to Send an ACK Segment

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

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

Note that there are several current practices that further lead to a reduced number of ACKs, including generic receive offload (GRO) [72], ACK compression, and ACK decimation [28].

3.9. Interfaces

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

3.9.1. User/TCP Interface

The following functional description of user commands to the TCP implementation is, at best, fictional, since every operating system

will have different facilities. Consequently, we must warn readers that different TCP implementations may have different user interfaces. However, all TCP implementations must provide a certain minimum set of services to guarantee that all TCP implementations can support the same protocol hierarchy. This section specifies the functional interfaces required of all TCP implementations.

Section 3.1 of [53] also identifies primitives provided by TCP and could be used as an additional reference for implementers.

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

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

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

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

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

3.9.1.1. Open

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

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

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

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

A TCP implementation that supports multiple concurrent connections MUST provide an OPEN call that will functionally allow an application to LISTEN on a port while a connection block with the same local port

is in SYN-SENT or SYN-RECEIVED state (MUST-42).

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

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

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

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

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

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

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

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

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

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

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

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

3.9.1.2. Send

Format: SEND (local connection name, buffer address, byte count, URGENT flag [, PUSH 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 PSH bit will be set in the last TCP segment created from the buffer.

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

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

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

New applications SHOULD NOT set the URGENT flag [39] due to implementation differences and middlebox issues (SHLD-13).

If the URGENT flag is set, segments sent to the destination TCP peer will have the urgent pointer set. The receiving TCP peer will signal the urgent condition to the receiving process if the urgent pointer indicates that data preceding the urgent pointer has not been consumed by the receiving process. The purpose of the URGENT flag is to stimulate the receiver to process the urgent data and to indicate to the receiver when all the currently known urgent data has been

received. The number of times the sending user's TCP implementation signals urgent will not necessarily be equal to the number of times the receiving user will be notified of the presence of urgent data.

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

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

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

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

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

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

3.9.1.3. Receive

Format: RECEIVE (local connection name, buffer address, byte count) -> byte count, URGENT flag [, PUSH flag]

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

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

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

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

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

3.9.1.4. Close

Format: CLOSE (local connection name)

This command causes the connection specified to be closed. If the connection is not open or the calling process is not authorized to use this connection, an error is returned. Closing connections is intended to be a graceful operation in the sense that outstanding SENDs will be transmitted (and retransmitted), as flow control permits, until all have been serviced. Thus, it should be acceptable to make several SEND calls, followed by a CLOSE, and expect all the data to be sent to the destination. It should also be clear that users should continue to RECEIVE on CLOSING connections since the remote peer may be trying to transmit the last of its data. Thus, CLOSE means "I have no more to send" but does not mean "I will not

receive any more." It may happen (if the user-level protocol is not well thought out) that the closing side is unable to get rid of all its data before timing out. In this event, CLOSE turns into ABORT, and the closing TCP peer gives up.

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

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

Close also implies push function.

3.9.1.5. Status

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

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

This command returns a data block containing the following information:

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

3.9.1.6. Abort

Format: ABORT (local connection name)

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

3.9.1.7. Flush

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

3.9.1.8. Asynchronous Reports

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

```
ERROR_REPORT(local connection name, reason, subreason)
```

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

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

* Excessive retransmissions (see Section 3.8.3)

* Urgent pointer advance (see Section 3.8.5)

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

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

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

change to the IP layer, when it sends segments on the connection (SHLD-22).

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

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

3.9.2. TCP/Lower-Level Interface

The TCP endpoint calls on a lower-level protocol module to actually send and receive information over a network. The two current standard Internet Protocol (IP) versions layered below TCP are IPv4 [1] and IPv6 [13].

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

Diffserv field: The IP header value for the Diffserv field is given by the user. This includes the bits of the Diffserv Codepoint (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 could not be delivered by the internet system within one minute. RFC 1122 updated RFC 793 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 (Sections 5.1, 5.3, and 6) [50]. Generally, an application SHOULD NOT change the Diffserv field value during the course of a connection (SHLD-23).

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

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

A TCP implementation MAY support the Timestamp (MAY-10) and Record Route (MAY-11) Options.

3.9.2.1. Source Routing

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

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

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

3.9.2.2. ICMP Messages

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

This applies to ICMPv6 in addition to IPv4 ICMP.

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

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

Soft Errors
: For IPv4 ICMP, these include: Destination Unreachable -- codes 0, 1, 5; Time Exceeded -- codes 0, 1; and Parameter Problem.

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

  Since these Unreachable messages indicate soft error conditions, a TCP implementation 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). [35] 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 [35], Section 4 describes widespread implementation behavior that treats soft errors as hard errors during connection establishment.

3.9.2.3. Source Address Validation

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

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

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

3.10. Event Processing

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

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

Events that occur:

- User Calls
  - OPEN
  - SEND
  - RECEIVE
  - CLOSE
  - ABORT
  - STATUS
- Arriving Segments

SEGMENT ARRIVES

Timeouts

USER TIMEOUT

RETRANSMISSION TIMEOUT

TIME-WAIT TIMEOUT

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

Error responses in this document are identified by character strings. For example, user commands referencing connections that do not exist receive "error: connection not open".

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

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

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

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

3.10.1. OPEN Call

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

* Create a new transmission control block (TCB) to hold connection state information. Fill in local socket identifier, remote socket, Diffserv field, security/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: Diffserv value 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 the OPEN call is active and the remote socket is specified, then change the connection from passive to active, select an ISS. Send a SYN segment, set SND.UNA to ISS, SND.NXT to ISS+1. Enter SYN-SENT state. Data associated with SEND may be sent with SYN segment or queued for transmission after entering ESTABLISHED state. The urgent bit if requested in the command must be sent with the data segments sent as a result of this command. If there is no room to queue the request, respond with "error: insufficient resources". If the remote socket was not specified, then return "error: remote socket unspecified".

SYN-SENT STATE

SYN-RECEIVED STATE

ESTABLISHED STATE

FIN-WAIT-1 STATE

FIN-WAIT-2 STATE

CLOSE-WAIT STATE

CLOSING STATE

LAST-ACK STATE

TIME-WAIT STATE

* Return "error: connection already exists".

3.10.2. SEND Call

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

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

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

LISTEN STATE

* If the remote socket is specified, then change the connection from passive to active, select an ISS. Send a SYN segment, set SND.UNA to ISS, SND.NXT to ISS+1. Enter SYN-SENT state. Data associated with SEND may be sent with SYN segment or queued for transmission after entering ESTABLISHED state. The urgent bit if requested in the command must be sent with the data segments sent as a result

of this command. If there is no room to queue the request, respond with "error: insufficient resources". If the remote socket was not specified, then return "error: remote socket unspecified".

SYN-SENT STATE

SYN-RECEIVED STATE

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

ESTABLISHED STATE

CLOSE-WAIT STATE

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

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

FIN-WAIT-1 STATE

FIN-WAIT-2 STATE

CLOSING STATE

LAST-ACK STATE

TIME-WAIT STATE

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

3.10.3. RECEIVE Call

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

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

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

LISTEN STATE

SYN-SENT STATE

SYN-RECEIVED STATE

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

ESTABLISHED STATE

FIN-WAIT-1 STATE

FIN-WAIT-2 STATE

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

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

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

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

CLOSE-WAIT STATE

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

CLOSING STATE

LAST-ACK STATE

TIME-WAIT STATE

* Return "error: connection closing".

3.10.4. CLOSE Call

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

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

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

LISTEN STATE

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

SYN-SENT STATE

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

SENDs, or RECEIVEs.

SYN-RECEIVED STATE

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

ESTABLISHED STATE

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

FIN-WAIT-1 STATE

FIN-WAIT-2 STATE

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

CLOSE-WAIT STATE

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

CLOSING STATE

LAST-ACK STATE

TIME-WAIT STATE

* Respond with "error: connection closing".

3.10.5. ABORT Call

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

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

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

LISTEN STATE

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

SYN-SENT STATE

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

SYN-RECEIVED STATE

ESTABLISHED STATE

FIN-WAIT-1 STATE

FIN-WAIT-2 STATE

CLOSE-WAIT STATE

* Send a reset segment:

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

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

CLOSING STATE

LAST-ACK STATE

TIME-WAIT STATE

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

3.10.6. STATUS Call

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

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

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

LISTEN STATE

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

SYN-SENT STATE

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

SYN-RECEIVED STATE

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

ESTABLISHED STATE

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

FIN-WAIT-1 STATE

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

FIN-WAIT-2 STATE

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

CLOSE-WAIT STATE

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

CLOSING STATE

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

LAST-ACK STATE

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

TIME-WAIT STATE

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

3.10.7. SEGMENT ARRIVES

3.10.7.1. CLOSED STATE

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

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

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

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

If the ACK bit is on,

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

Return.

3.10.7.2. LISTEN STATE

If the state is LISTEN, then

First, check for a RST:

- An incoming RST segment could not be valid since it could not have been sent in response to anything sent by this incarnation of the connection. An incoming RST should be ignored. Return.

Second, check for an ACK:

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

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

- Return.

Third, check for a SYN:

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

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

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

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

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

Fourth, other data or control:

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

3.10.7.3. SYN-SENT STATE

If the state is SYN-SENT, then

First, check the ACK bit:

- If the ACK bit is set,

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

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

  - o and discard the segment. Return.

  - o If SND.UNA < SEG.ACK =< SND.NXT, then the ACK is acceptable. Some deployed TCP code has used the check SEG.ACK == SND.NXT

(using "==" rather than "=<"), but this is not appropriate when the stack is capable of sending data on the SYN because the TCP peer may not accept and acknowledge all of the data on the SYN.

Second, check the RST bit:

- If the RST bit is set,

  - o A potential blind reset attack is described in RFC 5961 [9]. The mitigation described in that document has specific applicability explained therein, and is not a substitute for cryptographic protection (e.g., IPsec or TCP-AO). A TCP implementation that supports the mitigation described in RFC 5961 SHOULD first check that the sequence number exactly matches RCV.NXT prior to executing the action in the next paragraph.

  - o If the ACK was acceptable, then signal to 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:

  - o If there is an ACK,

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

  - o Otherwise,

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

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

Fourth, check the SYN bit:

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

- If the SYN bit is on and the security/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. Some TCP implementations

suppress sending this segment when the received segment contains data that will anyways generate an acknowledgment in the later processing steps, saving this extra acknowledgment of the SYN from being sent. If there are other controls or text in the segment, then continue processing at the sixth step under Section 3.10.7.4 where the URG bit is checked; otherwise, return.

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

  <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 [21] and is used in TCP Fast Open (TFO) [48], so is important for implementations and network devices to permit.

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

3.10.7.4. Other States

Otherwise,

First, check sequence number:

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

- TIME-WAIT STATE

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

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

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

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

Table 6: Segment Acceptability Tests

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

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

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

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

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

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

algorithm described in [40] for handling incoming SYN segments that utilizes timestamps rather than relying on the sequence number check described here. When the improved algorithm is implemented, the logic above is not applicable for incoming SYN segments with Timestamp Options, received on a connection in the TIME-WAIT state.

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

Second, check the RST bit:

- RFC 5961 [9], Section 3 describes a potential blind reset attack and optional mitigation approach. This does not provide a cryptographic protection (e.g., as in IPsec or TCP-AO) but can be applicable in situations described in RFC 5961. For stacks implementing the protection described in 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 acknowledgment (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 [99] contain additional considerations for ACK throttling in an implementation.

- SYN-RECEIVED STATE

  o If the RST bit is set,

    + If this connection was initiated with a passive OPEN (i.e., came from the LISTEN state), then return this connection to LISTEN state and return. The user need not be informed. If this connection was initiated with an active OPEN (i.e., came from SYN-SENT state), then the

connection was refused; signal the user "connection refused". In either case, the retransmission queue should be flushed. And in the active OPEN case, enter the CLOSED state and delete the TCB, and return.

- ESTABLISHED STATE

- FIN-WAIT-1 STATE

- FIN-WAIT-2 STATE

- CLOSE-WAIT STATE

  o 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 STATE

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

Third, check security:

- SYN-RECEIVED STATE

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

- ESTABLISHED STATE

- FIN-WAIT-1 STATE

- FIN-WAIT-2 STATE

- CLOSE-WAIT STATE

- CLOSING STATE

- LAST-ACK STATE

- TIME-WAIT STATE

  o 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.

Fourth, check the SYN bit:

- SYN-RECEIVED STATE

  - 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-1 STATE

- FIN-WAIT-2 STATE

- CLOSE-WAIT STATE

- CLOSING STATE

- LAST-ACK STATE

- TIME-WAIT STATE

  - If the SYN bit is set in these synchronized states, it may be either a legitimate new connection attempt (e.g., in the case of TIME-WAIT), an error where the connection should be reset, or the result of an attack attempt, as described in RFC 5961 [9]. For the TIME-WAIT state, new connections can be accepted if the Timestamp Option is used and meets expectations (per [40]). For all other cases, RFC 5961 provides a mitigation with applicability to some situations, though there are also alternatives that offer cryptographic protection (see Section 7). RFC 5961 recommends that in these synchronized states, if the SYN bit is set, irrespective of the sequence number, TCP endpoints MUST send a "challenge ACK" to the remote peer:

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

  - After sending the acknowledgment, TCP implementations MUST drop the unacceptable segment and stop processing further. Note that RFC 5961 and Errata ID 4772 [99] contain additional ACK throttling notes for an implementation.

  - For implementations that do not follow RFC 5961, the original behavior described in RFC 793 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.

- o 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,

  - o RFC 5961 [9], Section 5 describes a potential blind data injection attack, and mitigation that implementations MAY choose to include (MAY-12). TCP stacks that implement RFC 5961 MUST add an input check that the ACK value is acceptable only if it is in the range of ((SND.UNA - MAX.SND.WND) =< SEG.ACK =< SND.NXT). All incoming segments whose ACK value doesn't satisfy the above condition MUST be discarded and an ACK sent back. The new state variable MAX.SND.WND is defined as the largest window that the local sender has ever received from its peer (subject to window scaling) or may be hard-coded to a maximum permissible window value. When the ACK value is acceptable, the per-state processing below applies:

  - o SYN-RECEIVED STATE

    - + If SND.UNA < SEG.ACK =< SND.NXT, then enter ESTABLISHED state and continue processing with the 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.

  - o ESTABLISHED STATE

    - + If SND.UNA < SEG.ACK =< SND.NXT, then set SND.UNA <- SEG.ACK. Any segments on the retransmission queue that are thereby entirely acknowledged are removed. Users should receive positive acknowledgments for buffers that have been SENT and fully acknowledged (i.e., SEND buffer should be returned with "ok" response). If the ACK is a duplicate (SEG.ACK =< SND.UNA), it can be ignored. If

the ACK acks something not yet sent (SEG.ACK > SND.NXT), then send an ACK, drop the segment, and return.

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

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

o 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.

o 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.

o CLOSE-WAIT STATE

+ Do the same processing as for the ESTABLISHED state.

o 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.

o LAST-ACK STATE

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

o 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 STATE

  - 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 data to user RECEIVE buffers. Data from segments can be moved into buffers until either the buffer is full or the segment is empty. If the segment empties and carries a PUSH flag, then the user is informed, when the buffer is returned, that a PUSH has been received.

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

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

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

  - Please note the window management suggestions in Section 3.8.

  - Send an acknowledgment of the form:

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

  - This acknowledgment should be piggybacked on a segment being

transmitted if possible without incurring undue delay.

- CLOSE-WAIT STATE

- CLOSING STATE

- LAST-ACK STATE

- TIME-WAIT STATE

  o 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.

  o SYN-RECEIVED STATE

  o ESTABLISHED STATE

    + Enter the CLOSE-WAIT state.

  o FIN-WAIT-1 STATE

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

  o FIN-WAIT-2 STATE

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

  o CLOSE-WAIT STATE

    + Remain in the CLOSE-WAIT state.

  o CLOSING STATE

    + Remain in the CLOSING state.

  o LAST-ACK STATE

    + Remain in the LAST-ACK state.

  o TIME-WAIT STATE

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

and return.

3.10.8. Timeouts

USER TIMEOUT

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

RETRANSMISSION TIMEOUT

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

TIME-WAIT TIMEOUT

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

4. Glossary

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

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

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

Destination Address
: The network-layer address of the endpoint intended to receive a segment.

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

flush
: To remove all of the contents (data or segments) from a store (buffer or queue).

fragment

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

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

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

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

internet address
: A network-layer address.

internet datagram
: A unit of data exchanged between internet hosts, together with the internet header that allows the datagram to be routed from source to destination.

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

IP
: Internet Protocol. See [1] and [13].

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

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

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

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

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

MSL
: Maximum Segment Lifetime, the time a TCP segment can exist in

the internetwork system. Arbitrarily defined to be 2 minutes.

octet
An eight-bit byte.

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

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

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

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

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

RCV.NXT
receive next sequence number

RCV.UP
receive urgent pointer

RCV.WND
receive window

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

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

RST
A control bit (reset), occupying no sequence space, indicating that the receiver should delete the connection without further interaction. The receiver can determine, based on the sequence number and acknowledgment fields of the

incoming segment, whether it should honor the reset command or ignore it. In no case does receipt of a segment containing RST give rise to a RST in response.

SEG.ACK
: segment acknowledgment

SEG.LEN
: segment length

SEG.SEQ
: segment sequence

SEG.UP
: segment urgent pointer field

SEG.WND
: segment window field

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

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

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

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

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

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

SND.NXT
: send sequence

SND.UNA
: left sequence

SND.UP
send urgent pointer

SND.WL1
segment sequence number at last window update

SND.WL2
segment acknowledgment number at last window update

SND.WND
send window

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

Source Address
The network-layer address of the sending endpoint.

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

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

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

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

Type of Service
See "TOS".

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

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

5. Changes from RFC 793

This document obsoletes RFC 793 as well as RFCs 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 [73], 574 [74], 700 [75], 701 [76], 1283 [77], 1561 [78], 1562 [79], 1564 [80], 1571 [81], 1572 [82], 2297 [83], 2298 [84], 2748 [85], 2749 [86], 2934 [87], 3213 [88], 3300 [89], 3301 [90], 6222 [91]). Some errata were not applicable due to other changes (Errata IDs: 572 [92], 575 [93], 1565 [94], 1569 [95], 2296 [96], 3305 [97], 3602 [98]).

Changes to the specification of the urgent pointer described in RFCs 1011, 1122, and 6093 were incorporated. See RFC 6093 for detailed discussion of why these changes were necessary.

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

RFC 1011 [18] contains a number of comments about RFC 793, including some needed changes to the TCP specification. These are expanded in RFC 1122, which contains a collection of other changes and clarifications to RFC 793. The normative items impacting the protocol have been incorporated here, though some historically useful implementation advice and informative discussion from RFC 1122 is not included here. The present document, which is now the TCP specification rather than RFC 793, updates RFC 1011, and the comments noted in RFC 1011 have been incorporated.

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

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 the clarification it describes has been reflected within this base TCP specification.

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

6. IANA Considerations

In the "Transmission Control Protocol (TCP) Header Flags" registry, IANA has made several changes as 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 [54].

The "Bit" column has been renamed below as the "Bit Offset" column because 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 3 are the TCP segment Data Offset field, and not header flags.

IANA has added a column for "Assignment Notes".

IANA has assigned values as indicated below.

Bit Offset	Name	Reference	Assignment Notes
4	Reserved for future use	RFC 9293	
5	Reserved for future use	RFC 9293	
6	Reserved for future use	RFC 9293	
7	Reserved for future use	RFC 8311	Previously used by Historic RFC 3540 as NS (Nonce Sum).
8	CWR (Congestion Window Reduced)	RFC 3168	
9	ECE (ECN-Echo)	RFC 3168	
10	Urgent pointer field is significant (URG)	RFC 9293	
11	Acknowledgment field is significant (ACK)	RFC 9293	
12	Push function	RFC 9293	

	(PSH)			
13	Reset the connection (RST)	RFC 9293		
14	Synchronize sequence numbers (SYN)	RFC 9293		
15	No more data from sender (FIN)	RFC 9293		

Table 7: TCP Header Flags

The "TCP Header Flags" registry has also been moved to a subregistry under the global "Transmission Control Protocol (TCP) Parameters" registry <https://www.iana.org/assignments/tcp-parameters/>.

The registry's Registration Procedure remains Standards Action, but the Reference has been updated to this document, and the Note has been removed.

7. Security and Privacy Considerations

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

In order to fully provide confidentiality, integrity protection, and authentication for TCP connections (including their control flags), IPsec is the only current effective method. For integrity protection and authentication, the TCP Authentication Option (TCP-AO) [38] is available, with a proposed extension to also provide confidentiality for the segment payload. Other methods discussed in this section may provide confidentiality or integrity protection for the payload, but for the TCP header only cover either a subset of the fields (e.g., tcpcrypt [57]) or none at all (e.g., TLS). Other security features that have been added to TCP (e.g., ISN generation, sequence number checks, and others) are only capable of partially hindering attacks.

Applications using long-lived TCP flows have been vulnerable to attacks that exploit the processing of control flags described in earlier TCP specifications [33]. 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 or for the TCP headers.

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

The TCP Roadmap [49] 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 [39] for discussion of security considerations related to the urgent pointer field, which also discourages new applications from using the urgent pointer.

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) [29] that act as mitigations to these attacks.

Other attacks are focused on exhausting the resources of a TCP server. Examples include SYN flooding [32] or wasting resources on non-progressing connections [41]. Operating systems commonly implement mitigations for these attacks. Some common defenses also utilize proxies, stateful firewalls, and other technologies outside the end-host TCP implementation.

The concept of a protocol's "wire image" is described in RFC 8546 [56], which describes how TCP's cleartext headers expose more metadata to nodes on the path than is strictly required to route the packets to their destination. On-path adversaries may be able to leverage this metadata. Lessons learned in this respect from TCP have been applied in the design of newer transports like QUIC [60]. Additionally, based partly on experiences with TCP and its extensions, there are considerations that might be applicable for future TCP extensions and other transports that the IETF has documented in RFC 9065 [61], along with IAB recommendations in RFC 8558 [58] and [67].

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

Since ICMP message processing also can interact with TCP connections,

there is potential for ICMP-based attacks against TCP connections. These are discussed in RFC 5927 [100], along with mitigations that have been implemented.

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

[4] Nichols, K., Blake, S., Baker, F., and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", RFC 2474, DOI 10.17487/RFC2474, December 1998, <https://www.rfc-editor.org/info/rfc2474>.

[5] Floyd, S., "Congestion Control Principles", BCP 41, RFC 2914, DOI 10.17487/RFC2914, September 2000, <https://www.rfc-editor.org/info/rfc2914>.

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

[7] Floyd, S. and M. Allman, "Specifying New Congestion Control Algorithms", BCP 133, RFC 5033, DOI 10.17487/RFC5033, August 2007, <https://www.rfc-editor.org/info/rfc5033>.

[8] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <https://www.rfc-editor.org/info/rfc5681>.

[9] Ramaiah, A., Stewart, R., and M. Dalal, "Improving TCP's Robustness to Blind In-Window Attacks", RFC 5961, DOI 10.17487/RFC5961, August 2010, <https://www.rfc-editor.org/info/rfc5961>.

[10] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, DOI 10.17487/RFC6298, June 2011, <https://www.rfc-editor.org/info/rfc6298>.

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

RFC 6633, DOI 10.17487/RFC6633, May 2012,
<https://www.rfc-editor.org/info/rfc6633>.

[12] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[13] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", STD 86, RFC 8200, DOI 10.17487/RFC8200, July 2017, <https://www.rfc-editor.org/info/rfc8200>.

[14] McCann, J., Deering, S., Mogul, J., and R. Hinden, Ed., "Path MTU Discovery for IP version 6", STD 87, RFC 8201, DOI 10.17487/RFC8201, July 2017, <https://www.rfc-editor.org/info/rfc8201>.

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

8.2. Informative References

[16] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

[17] Nagle, J., "Congestion Control in IP/TCP Internetworks", RFC 896, DOI 10.17487/RFC0896, January 1984, <https://www.rfc-editor.org/info/rfc896>.

[18] Reynolds, J. and J. Postel, "Official Internet protocols", RFC 1011, DOI 10.17487/RFC1011, May 1987, <https://www.rfc-editor.org/info/rfc1011>.

[19] 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>.

[20] 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>.

[21] 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>.

[22] Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, "TCP Selective Acknowledgment Options", RFC 2018, DOI 10.17487/RFC2018, October 1996, <https://www.rfc-editor.org/info/rfc2018>.

[23] 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>.

[24] Borman, D., Deering, S., and R. Hinden, "IPv6 Jumbograms", RFC 2675, DOI 10.17487/RFC2675, August 1999, <https://www.rfc-editor.org/info/rfc2675>.

[25] 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>.

[26] Floyd, S., Mahdavi, J., Mathis, M., and M. Podolsky, "An Extension to the Selective Acknowledgement (SACK) Option for TCP", RFC 2883, DOI 10.17487/RFC2883, July 2000, <https://www.rfc-editor.org/info/rfc2883>.

[27] Lahey, K., "TCP Problems with Path MTU Discovery", RFC 2923, DOI 10.17487/RFC2923, September 2000, <https://www.rfc-editor.org/info/rfc2923>.

[28] 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>.

[29] 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>.

[30] 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>.

[31] 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>.

[32] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", RFC 4987, DOI 10.17487/RFC4987, August 2007, <https://www.rfc-editor.org/info/rfc4987>.

[33] Touch, J., "Defending TCP Against Spoofing Attacks", RFC 4953, DOI 10.17487/RFC4953, July 2007, <https://www.rfc-editor.org/info/rfc4953>.

[34] 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>.

[35] Gont, F., "TCP's Reaction to Soft Errors", RFC 5461, DOI 10.17487/RFC5461, February 2009, <https://www.rfc-editor.org/info/rfc5461>.

[36] 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>.

[37] 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>.

[38] 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>.

[39] 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>.

[40] 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>.

[41] 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>.

[42] 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>.

[43] Borman, D., "TCP Options and Maximum Segment Size (MSS)", RFC 6691, DOI 10.17487/RFC6691, July 2012, <https://www.rfc-editor.org/info/rfc6691>.

[44] Touch, J., "Updated Specification of the IPv4 ID Field", RFC 6864, DOI 10.17487/RFC6864, February 2013, <https://www.rfc-editor.org/info/rfc6864>.

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

[46] McPherson, D., Oran, D., Thaler, D., and E. Osterweil, "Architectural Considerations of IP Anycast", RFC 7094, DOI 10.17487/RFC7094, January 2014, <https://www.rfc-editor.org/info/rfc7094>.

[47] 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>.

[48] 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>.

[49] 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>.

[50] 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>.

[51] 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>.

[52] 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>.

[53] 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>.

[54] Black, D., "Relaxing Restrictions on Explicit Congestion Notification (ECN) Experimentation", RFC 8311, DOI 10.17487/RFC8311, January 2018, <https://www.rfc-editor.org/info/rfc8311>.

[55] Chown, T., Loughney, J., and T. Winters, "IPv6 Node Requirements", BCP 220, RFC 8504, DOI 10.17487/RFC8504, January 2019, <https://www.rfc-editor.org/info/rfc8504>.

[56] Trammell, B. and M. Kuehlewind, "The Wire Image of a Network Protocol", RFC 8546, DOI 10.17487/RFC8546, April 2019, <https://www.rfc-editor.org/info/rfc8546>.

[57] Bittau, A., Giffin, D., Handley, M., Mazieres, D., Slack, Q., and E. Smith, "Cryptographic Protection of TCP Streams (tcpcrypt)", RFC 8548, DOI 10.17487/RFC8548, May 2019, <https://www.rfc-editor.org/info/rfc8548>.

[58] Hardie, T., Ed., "Transport Protocol Path Signals", RFC 8558, DOI 10.17487/RFC8558, April 2019, <https://www.rfc-editor.org/info/rfc8558>.

[59] Ford, A., Raiciu, C., Handley, M., Bonaventure, O., and C. Paasch, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 8684, DOI 10.17487/RFC8684, March 2020, <https://www.rfc-editor.org/info/rfc8684>.

[60] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based

Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/info/rfc9000>.

[61] Fairhurst, G. and C. Perkins, "Considerations around Transport Header Confidentiality, Network Operations, and the Evolution of Internet Transport Protocols", RFC 9065, DOI 10.17487/RFC9065, July 2021, <https://www.rfc-editor.org/info/rfc9065>.

[62] IANA, "Transmission Control Protocol (TCP) Parameters", <https://www.iana.org/assignments/tcp-parameters/>.

[63] Gont, F., "Processing of IP Security/Compartment and Precedence Information by TCP", Work in Progress, Internet-Draft, draft-gont-tcpm-tcp-seccomp-prec-00, 29 March 2012, <https://datatracker.ietf.org/doc/html/draft-gont-tcpm-tcp-seccomp-prec-00>.

[64] Gont, F. and D. Borman, "On the Validation of TCP Sequence Numbers", Work in Progress, Internet-Draft, draft-gont-tcpm-tcp-seq-validation-04, 11 March 2019, <https://datatracker.ietf.org/doc/html/draft-gont-tcpm-tcp-seq-validation-04>.

[65] Touch, J. and W. M. Eddy, "TCP Extended Data Offset Option", Work in Progress, Internet-Draft, draft-ietf-tcpm-tcp-edo-12, 15 April 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-tcpm-tcp-edo-12>.

[66] McQuistin, S., Band, V., Jacob, D., and C. Perkins, "Describing Protocol Data Units with Augmented Packet Header Diagrams", Work in Progress, Internet-Draft, draft-mcquistin-augmented-ascii-diagrams-10, 7 March 2022, <https://datatracker.ietf.org/doc/html/draft-mcquistin-augmented-ascii-diagrams-10>.

[67] Thomson, M. and T. Pauly, "Long-Term Viability of Protocol Extension Mechanisms", RFC 9170, DOI 10.17487/RFC9170, December 2021, <https://www.rfc-editor.org/info/rfc9170>.

[68] Minshall, G., "A Suggested Modification to Nagle's Algorithm", Work in Progress, Internet-Draft, draft-minshall-nagle-01, 18 June 1999, <https://datatracker.ietf.org/doc/html/draft-minshall-nagle-01>.

[69] Dalal, Y. and C. Sunshine, "Connection Management in Transport Protocols", Computer Networks, Vol. 2, No. 6, pp. 454-473, DOI 10.1016/0376-5075(78)90053-3, December 1978, <https://doi.org/10.1016/0376-5075(78)90053-3>.

[70] Faber, T., Touch, J., and W. Yui, "The TIME-WAIT state in TCP and Its Effect on Busy Servers", Proceedings of IEEE INFOCOM, pp. 1573-1583, DOI 10.1109/INFCOM.1999.752180,

March 1999, <https://doi.org/10.1109/INFCOM.1999.752180>.

[71] Postel, J., "Comments on Action Items from the January Meeting", IEN 177, March 1981, <https://www.rfc-editor.org/ien/ien177.txt>.

[72] "Segmentation Offloads", The Linux Kernel Documentation, <https://www.kernel.org/doc/html/latest/networking/segmentation-offloads.html>.

[73] RFC Errata, Erratum ID 573, RFC 793, <https://www.rfc-editor.org/errata/eid573>.

[74] RFC Errata, Erratum ID 574, RFC 793, <https://www.rfc-editor.org/errata/eid574>.

[75] RFC Errata, Erratum ID 700, RFC 793, <https://www.rfc-editor.org/errata/eid700>.

[76] RFC Errata, Erratum ID 701, RFC 793, <https://www.rfc-editor.org/errata/eid701>.

[77] RFC Errata, Erratum ID 1283, RFC 793, <https://www.rfc-editor.org/errata/eid1283>.

[78] RFC Errata, Erratum ID 1561, RFC 793, <https://www.rfc-editor.org/errata/eid1561>.

[79] RFC Errata, Erratum ID 1562, RFC 793, <https://www.rfc-editor.org/errata/eid1562>.

[80] RFC Errata, Erratum ID 1564, RFC 793, <https://www.rfc-editor.org/errata/eid1564>.

[81] RFC Errata, Erratum ID 1571, RFC 793, <https://www.rfc-editor.org/errata/eid1571>.

[82] RFC Errata, Erratum ID 1572, RFC 793, <https://www.rfc-editor.org/errata/eid1572>.

[83] RFC Errata, Erratum ID 2297, RFC 793, <https://www.rfc-editor.org/errata/eid2297>.

[84] RFC Errata, Erratum ID 2298, RFC 793, <https://www.rfc-editor.org/errata/eid2298>.

[85] RFC Errata, Erratum ID 2748, RFC 793, <https://www.rfc-editor.org/errata/eid2748>.

[86] RFC Errata, Erratum ID 2749, RFC 793, <https://www.rfc-editor.org/errata/eid2749>.

[87] RFC Errata, Erratum ID 2934, RFC 793, <https://www.rfc-editor.org/errata/eid2934>.

[88] RFC Errata, Erratum ID 3213, RFC 793,

<https://www.rfc-editor.org/errata/eid3213>.

[89] RFC Errata, Erratum ID 3300, RFC 793, <https://www.rfc-editor.org/errata/eid3300>.

[90] RFC Errata, Erratum ID 3301, RFC 793, <https://www.rfc-editor.org/errata/eid3301>.

[91] RFC Errata, Erratum ID 6222, RFC 793, <https://www.rfc-editor.org/errata/eid6222>.

[92] RFC Errata, Erratum ID 572, RFC 793, <https://www.rfc-editor.org/errata/eid572>.

[93] RFC Errata, Erratum ID 575, RFC 793, <https://www.rfc-editor.org/errata/eid575>.

[94] RFC Errata, Erratum ID 1565, RFC 793, <https://www.rfc-editor.org/errata/eid1565>.

[95] RFC Errata, Erratum ID 1569, RFC 793, <https://www.rfc-editor.org/errata/eid1569>.

[96] RFC Errata, Erratum ID 2296, RFC 793, <https://www.rfc-editor.org/errata/eid2296>.

[97] RFC Errata, Erratum ID 3305, RFC 793, <https://www.rfc-editor.org/errata/eid3305>.

[98] RFC Errata, Erratum ID 3602, RFC 793, <https://www.rfc-editor.org/errata/eid3602>.

[99] RFC Errata, Erratum ID 4772, RFC 5961, <https://www.rfc-editor.org/errata/eid4772>.

[100] Gont, F., "ICMP Attacks against TCP", RFC 5927, DOI 10.17487/RFC5927, July 2010, <https://www.rfc-editor.org/info/rfc5927>.

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 (TOS) field. It was modified in [20] and then obsoleted by the definition of Differentiated Services (Diffserv) [4]. Setting and conveying TOS between the network layer, TCP implementation, and applications is obsolete and is replaced by Diffserv in the current TCP specification.

RFC 793 required checking the IP security compartment and precedence on incoming TCP segments for consistency within a connection and with application requests. Each of these aspects of IP have become outdated, without specific updates to RFC 793. The issues with precedence were fixed by [25], 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 Multi-Level Secure (MLS) systems is not as apparent in the IETF currently.

Resetting connections when incoming packets do not meet expected security compartment or precedence expectations has been recognized as a possible attack vector [63], and there has been discussion about amending the TCP specification to prevent connections from being aborted due to nonmatching 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 TCP specification defined by RFCs 793 and 1122 included 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 to 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 (Sections 5.1, 5.3, and 6) [50].

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 [25], 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 in 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, which was later obsoleted by RFC 1108. The Commercial IP Security Option (CIPSO) is defined in FIPS-188 (withdrawn by NIST in 2015) and is supported by some vendors and operating systems. RFC 1108 is now Historic, though RFC 791 itself has not been updated to remove the IP Security Option. For IPv6, a similar option (Common Architecture Label IPv6 Security Option (CALIPSO)) has been defined [36]. RFC 793 includes logic that includes the IP security/compartment information in treatment of TCP segments. References to the IP "security/compartment" in this document may be relevant for Multi-Level Secure (MLS) system implementers but can be ignored for non-MLS implementations, consistent with running code on the Internet. See Appendix A.1 for further discussion. Note that RFC 5570 describes some MLS networking

scenarios where IPSO, CIPSO, or CALIPSO may be used. In these special cases, TCP implementers should see Section 7.3.1 of RFC 5570 and follow the guidance in that document.

A.2. Sequence Number Validation

There are cases where the TCP sequence number validation rules can prevent ACK fields from being processed. This can result in connection issues, as described in [64], 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 rarely occur. 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 [64].

A.3. Nagle Modification

In common operating systems, both the Nagle algorithm and delayed acknowledgments 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 acknowledgments can result in poor application performance. A modification to the Nagle algorithm is described in [68] that improves the situation for these applications.

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

A.4. Low Watermark Settings

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

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

Appendix B. TCP Requirement Summary

This section is adapted from RFC 1122.

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

but that PLPMTUD is recommended.

Feature	ReqID	MUST	SHOULD	MAY	SHOULD NOT	MUST NOT
PUSH flag						
Aggregate or queue un-pushed data	MAY-16			X		
Sender collapse successive PSH bits	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^1 of PSH	MAY-17			X		
Send max size segment when possible	SHLD-28		X			
Window						
Treat as unsigned number	MUST-1	X				
Handle as 32-bit number	REC-1		X			
Shrink window from right	SHLD-14				X	
* Send new data when window shrinks	SHLD-15				X	
* Retransmit old unacked data within window	SHLD-16		X			

* Time out conn for data past right edge	SHLD-17				X	
Robust against shrinking window	MUST-34	X				
Receiver's window closed indefinitely	MAY-8			X		
Use standard probing logic	MUST-35	X				
Sender probe zero window	MUST-36	X				
* First probe after RTO	SHLD-29		X			
* Exponential backoff	SHLD-30		X			
Allow window stay zero indefinitely	MUST-37	X				
Retransmit old data beyond SND.UNA+SND.WND	MAY-7			X		
Process RST and URG even with zero window	MUST-66	X				
Urgent Data						
Include support for urgent pointer	MUST-30	X				
Pointer indicates first non-urgent octet	MUST-62	X				
Arbitrary length urgent data sequence	MUST-31	X				
Inform ALP^1 asynchronously of urgent data	MUST-32	X				

ALP^1 can learn if/how much urgent data Q'd	MUST-33	X				
ALP employ the urgent mechanism	SHLD-13				X	
TCP Options						
Support the mandatory option set	MUST-4	X				
Receive TCP Option in any segment	MUST-5	X				
Ignore unsupported options	MUST-6	X				
Include length for all options except EOL+NOP	MUST-68	X				
Cope with illegal option length	MUST-7	X				
Process options regardless of word alignment	MUST-64	X				
Implement sending & receiving MSS Option	MUST-14	X				
IPv4 Send MSS Option unless 536	SHLD-5		X			
IPv6 Send MSS Option unless 1220	SHLD-5		X			
Send MSS Option always	MAY-3			X		
IPv4 Send-MSS default is 536	MUST-15	X				
IPv6 Send-MSS default is 1220	MUST-15	X				

Calculate effective send seg size	MUST-16	X				
MSS accounts for varying MTU	SHLD-6		X			
MSS not sent on non-SYN segments	MUST-65					X
MSS value based on MMS_R	MUST-67	X				
Pad with zero	MUST-69	X				
TCP Checksums						
Sender compute checksum	MUST-2	X				
Receiver check checksum	MUST-3	X				
ISN Selection						
Include a clock-driven ISN generator component	MUST-8	X				
Secure ISN generator with a PRF component	SHLD-1		X			
PRF computable from outside the host	MUST-9					X
Opening Connections						
Support simultaneous open attempts	MUST-10	X				
SYN-RECEIVED remembers last state	MUST-11	X				
Passive OPEN call interfere with others	MUST-41					X
Function: simultaneously	MUST-42	X				

LISTENs for same port						
Ask IP for src address for SYN if necessary	MUST-44	X				
* Otherwise, use local addr of connection	MUST-45	X				
OPEN to broadcast/ multicast IP address	MUST-46					X
Silently discard seg to bcast/mcast addr	MUST-57	X				
Closing Connections						
RST can contain data	SHLD-2		X			
Inform application of aborted conn	MUST-12	X				
Half-duplex close connections	MAY-1			X		
* Send RST to indicate data lost	SHLD-3		X			
In TIME-WAIT state for 2MSL seconds	MUST-13	X				
* Accept SYN from TIME-WAIT state	MAY-2			X		
* Use Timestamps to reduce TIME-WAIT	SHLD-4		X			
Retransmissions						
Implement exponential	MUST-19	X				

backoff, slow start, and congestion avoidance							
Retransmit with same IP identity	MAY-4			X			
Karn's algorithm	MUST-18	X					
Generating ACKs							
Aggregate whenever possible	MUST-58	X					
Queue out-of-order segments	SHLD-31		X				
Process all Q'd before send ACK	MUST-59	X					
Send ACK for out-of-order segment	MAY-13			X			
Delayed ACKs	SHLD-18		X				
* Delay < 0.5 seconds	MUST-40	X					
* Every 2nd full-sized segment or 2*RMSS ACK'd	SHLD-19		X				
Receiver SWS-Avoidance Algorithm	MUST-39	X					
Sending Data							
Configurable TTL	MUST-49	X					
Sender SWS-Avoidance Algorithm	MUST-38	X					
Nagle algorithm	SHLD-7		X				
* Application can disable Nagle	MUST-17	X					

algorithm						
Connection Failures						
Negative advice to IP on R1 retransmissions	MUST-20	X				
Close connection on R2 retransmissions	MUST-20	X				
ALP^1 can set R2	MUST-21	X				
Inform ALP of R1<=retxs<R2	SHLD-9		X			
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						
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 ACKs	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					
Timestamp support	MAY-10		X				
Record Route support	MAY-11		X				
Source Route:							
* ALP^1 can specify	MUST-51	X					
* Overrides src route in datagram	MUST-52	X					
* Build return route from src route	MUST-53	X					
* Later src route overrides	SHLD-24		X				
Receiving ICMP Messages from IP							
Receiving ICMP messages from IP	MUST-54	X					
* Dest Unreach (0,1,5) => inform ALP	SHLD-25	X					
* Abort on Dest Unreach (0,1,5)	MUST-56					X	
* Dest Unreach (2-4) => abort conn	SHLD-26		X				

* Source Quench => silent discard	MUST-55	X					
* Abort on Time Exceeded	MUST-56						X
* Abort on Param Problem	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	MAY-9			X			

Diffserv field up to ALP							
FLUSH call	MAY-14			X			
Optional local IP addr param in OPEN	MUST-43	X					
RFC 5961 Support							
Implement data injection protection	MAY-12			X			
Explicit Congestion Notification							
Support ECN	SHLD-8		X				
Alternative Congestion Control							
Implement alternative conformant algorithm(s)	MAY-18			X			

Table 8: TCP Requirements Summary

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

Acknowledgments

This document is largely a revision of RFC 793, of which Jon Postel was the editor. 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 Tüxen

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

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

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

This document includes content from errata that were reported by (listed chronologically): Yin Shuming, Bob Braden, Morris M. Keesan, Pei-chun Cheng, Constantin Hagemeier, Vishwas Manral, Mykyta Yevstifeyev, EungJun Yi, Botong Huang, Charles Deng, Merlin Buge.

Author's Address

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