TCPM
Internet-Draft
Intended Status: Experimental
File: draft-borman-tcpm-tcp4way-01.txt
Expires: September 9, 2015

D. Borman
Quantum Corporation

March 9, 2015

TCP Four-Way Handshake

Abstract

One of the limitations of TCP is that it has limited space for TCP options, only 40 bytes. Many mechanisms have been proposed for for extending the TCP option space, but the biggest challenge has been to get additional option space in the initial SYN packet.

This memo presents a optional four-way TCP handshake to allow extended option space to be used in SYN packets in both directions.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current.

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

This Internet-Draft will expire on September 9, 2015.

Copyright

Table of Contents

1. Introduction 3
2. Terminology 3
3. Motivation For this Approach 3
4. TCP Four-Way Handshake 4
4.1. Overview 5
4.2. Changes to the TCP state diagram 6
4.3. Three-Way or Four-Way Handshake? 7
4.3.1. Using the 4WAY Bit 7
4.3.2. Non Four-Way Client Sets 4WAY Bit 7
4.3.3. Non Four-Way Server Sets 4WAY Bit 8
4.4. PRE-ESTABLISHED State 8
5. Negotiating Non-directional vs. Directional TCP Options 8
5.1. Caching EDO support 9
6. TCP Connection State Diagram 9
7. Using TCP Options in the Four-Way Handshake 12
7.1. New TCP Options in the SYN/ACK 12
7.2. Handling Unknown TCP Options 13
7.2.1. Unknown Option Option 13
7.3. TCP options 14
7.3.1. RFC defined options 14
7.3.1.1. End of Option List 14
7.3.1.2. No-Operation 14
7.3.1.3. Maximum Segment Size 14
7.3.1.4. Window Scale 14
7.3.1.5. SACK Permitted 14
7.3.1.6. SACK 14
7.3.1.7. Timestamps 15
7.3.1.8. MD5 Signature Option 15
7.3.1.9. Quick-Start Response 15
7.3.1.10. User Timeout Option 15
7.3.1.11. TCP Authentication Options (TCP-AO) 15
7.3.1.12. Multipath TCP (MPTCP) 15
7.3.1.13. RFC3692-style Experiment 15
7.3.1.14. TCP Fast Open Option 15
7.3.1.15. T/TCP Options 16
8. IANA Considerations 16
9. Security Considerations 16
10. References 16
10.1. Normative References 16
10.2. Informative References 17
Appendix A. First Response of the Four-Way Handshake 19
Appendix B. Communicating Four-Way Handshake Support 20
Acknowledgments 20
Contributors 20
Author’s Address 21

1. Introduction

The TCP packet format has 40 bytes for adding TCP options. The most common method to extend TCP is to define new options, but the limited TCP option space can make that difficult as the number of potential options grow. Support for various TCP options is typically negotiated during the three-way handshake, in the packets that contain the SYN. If both sides send and receive a given option in a packet with the SYN bit set, then both sides know that the option is supported. Examples of this are the Window Scale and Timestamps options [RFC7323]. Note that RFC 7323 is not clear on this point, its description is Three-Way handshake centric, stating that the Timestamps option is enabled by its presence in the <SYN> and <SYN,ACK> packets, rather than the more general definition that the option is enabled by both sending and receiving the option in a packet that contains the SYN bit. It is a subtle difference that doesn't matter for the Three-Way handshake, but is important for a simultaneous open and the Four-Way handshake.

The majority of TCP sessions begin with three-way handshake, the exception to that is a simultaneous open.

The ideas presented in this memo were first hinted at in a message to the TCPM mailing list [Borman14].

2. Terminology

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

3. Motivation For this Approach

The problem of expanding the TCP option space in the initial SYN packet has vexed designers for years. The main issue is maintaining compatibility with legacy TCP implementations, which don't understand the expanded TCP option space. When the initial SYN is sent, there is no knowledge as to whether or not the remote side can understand the extended option space. Various approaches have been considered:

1) Send dual SYNs, with and without the extended options, and arrange that the extended SYN will be considered invalid and dropped by legacy implementations. [Yourtchenko11] [Briscoe14]
2) Send an additional out of band packet along with the SYN to contain additional options. [Touch14]
3) Send additional options that didn't fit into the SYN in additional packets using a new TCP option. [Eddy08]
4) Send an initial SYN with extended options that a legacy server will fail, and then fall back to a new SYN without extended

options. [Kohler04]

[Ramaiah12] contains additional analysis of proposed ways to expand the TCP option space.

Expanding the TCP option space in the initial SYN is a case of the more general issue: How can you change the fixed TCP header in the initial SYN packet and still maintain compatibility with legacy implementations? The TCP Window Scale option [RFC7323] redefined the Window field, but only in non-SYN packets. In the case of expanding the TCP option space, it involves redefining or overriding the Data Offset (DO) field.

The most straight forward method for dealing with modifying the initial SYN packet is to add an initial packet exchange so that the client can find out what the server supports, and then it knows, for example, if the server supports extended TCP option space. The problem with this approach is that it adds an additional RTT to connection startup, and most people are looking for ways to shorten, not lengthen, the initial connection setup, for example "TCP Fast Open" [TFO]. Though to be clear, the extra RTT is really not a concern about connection setup, but about when data can be first delivered to the application.

An alternative is to send the additional data in the initial SYN such that a legacy TCP will ignore it. This is most commonly done by sending the information in a TCP option, which legacy TCP would ignore. But the TCP option space is only 40 bytes, and by definition an expanded TCP option space won't fit in the legacy TCP option space. So, the additional data needs to be sent by some other mechanism, e.g. in a second SYN or in an additional non-SYN packet. Challenges with this approach include the SYNs being routed to different destination machines, the order of the packets being reversed, as well as a server needing to wait some amount of time to decide whether or not the additional packet will be arriving.

The goal of this proposal is to integrate discovery of server capabilities into the connection setup, while still allowing for data to be delivered in a timely manner.

4. TCP Four-Way Handshake

The TCP Four-Way handshake extends the traditional Three-Way handshake by changing the clients final ACK to a SYN/ACK, and adding a final ACK from the Server. This gives the client a second packet with the SYN bit set in which it can make use of the EDO option to send TCP options that didn't fit into the original SYN, with some limitations.

4.1. Overview

For a connection with ISS (Initial Send Sequence) values of ISSA from the client and ISSB from the server, the normal three-way TCP handshake is:

```
    Enter SYN-SENT
    SYN(seq=ISSA) ->
                                  Enter SYN-RECEIVED
                               <- SYN(seq=ISSB)/ACK(ISSA)
    Enter ESTABLISHED
    ACK(ISSB) ->
                                  Enter ESTABLISHED
```

A simultaneous open is:

```
    Enter SYN-SENT                Enter SYN-SENT
    SYN(seq=ISSA) ->           <- SYN(seq=ISSB)

    Enter SYN-RECEIVED            Enter SYN-RECEIVED
    SYN(seq=ISSA)/ACK(ISSB) ->  <- SYN(seq=ISSB)/ACK(ISSA)

    Enter ESTABLISHED             Enter ESTABLISHED
```

See [RFC793] page 68 and [RFC1122] page 86.

The normal scenario for the proposed four-way handshake is:

```
    Enter SYN-SENT
    SYN(seq=ISSA) ->
                                  Enter SYN-SENT
                               <- SYN(seq=ISSB)/ACK(ISSA)
    Enter PRE-ESTABLISHED
    SYN(seq=ISSA)/ACK(ISSB) ->
                                  Enter ESTABLISHED
                               <- ACK(ISSA)
    Enter ESTABLISHED
```

There are other options for the initial server response in the four-way handshake. Those are discussed in Appendix A as well as the reasons they weren't chosen.

4.2. Changes to the TCP state diagram

The changes can be described entirely as new new state transitions and some additional decisions:

```
LISTEN -> rcv SYN,
    if (allow4way)
        passive4way=1, snd SYN,ACK -> SYN-SENT
    else
        passive4way=0, snd SND,ACK -> SYN-RCVD
SYN-SENT -> rcv ACK
    if (passive4way == 1)
        -> ESTABLISHED
    else
        normal error processing

CLOSED -> active OPEN, create TCB, snd SYN,
                       active4way=1 -> SYN-SENT

SYN-SENT -> rcv SYN,ACK
    if (active4way == 1 && (continue4way))
        snd SYN,ACK -> PRE-ESTABLISHED
    else
        snd ACK -> ESTABLISHED
```

The "allow4way" and "continue4way" decisions are based on the contents of the inbound packet.

Instead of overloading the SYN-SENT state and burying the decisions in the existing LISTEN and SYN-SENT states, the state diagram is expanded with two new states, SYN-ACK-SENT and PRE-ESTABLISHED, and two transitional states, ALLOW-4WAY and CONTINUE-4WAY. These are transitional because once entered, an immediate decision is made and then they are exited to a new state.

The LISTEN -> SYN-RCVD transition is replaced by:

```
LISTEN -> rcv SYN -> ALLOW-4WAY

ALLOW-4WAY(YES) -> snd SYN,ACK -> SYN-ACK-SENT
ALLOW-4WAY(NO) -> snd SYN,ACK -> SYN-RCVD

SYN-ACK-SENT -> rcv SYN,ACK, snd ACK -> ESTABLISHED
SYN-ACK-SENT -> rcv ACK of SYN, x -> ESTABLISHED
```

and the SYN-SENT -> ESTABLISHED transition is replaced by:

```
SYN-SENT -> rcv SYN,ACK -> CONTINUE-4WAY

CONTINUE-4WAY(YES) -> snd SYN,ACK -> PRE-ESTABLISHED
CONTINUE-4WAY(NO) -> snd ACK -> ESTABLISHED

PRE-ESTABLISHED -> rcv RST -> LISTEN
PRE-ESTABLISHED -> rcv ACK of SYN -> ESTABLISHED
PRE-ESTABLISHED -> CLOSE/snd FIN -> FIN WAIT-1
```

4.3. Three-Way or Four-Way Handshake?

There are two new decision points for for handling a four-way handshake. First, when a connection in LISTEN state receives a SYN packet, it has to decide based on the contents of that packet whether or not the remote side understands the four-way handshake. This is accomplished through the allocation of one of the unused bits in the TCP header, the 4WAY bit.

> Note: There are other ways to convey support for the four-way handshake instead of using an unused header bit. These are discussed in Appendix B.

4.3.1. Using the 4WAY Bit

The client sets the 4WAY bit in the initial SYN. If the server receives a 4WAY bit in the initial SYN, then it will set the 4WAY bit in the SYN/ACK. If the client receives a SYN/ACK without the 4WAY bit set, it proceeds with the normal three-way handshake. If it receives a SYN/ACK with the 4WAY bit set, then based on the options in the SYN/ACK it can chose to either proceed with the normal three-way handshake, or to continue with the four-way handshake.

If a packet is received with the 4WAY bit set, but not the SYN bit, the 4WAY bit is ignored. When sending a packet without the SYN bit set, the 4WAY bit must not be set.

[RFC3168] notes TCP interoperability issues with the CWR and ECE bits, but the 4WAY bit does not have the same issues.

4.3.2. Non Four-Way Client Sets 4WAY Bit

In this case, the server might enter SYN-ACK-SENT state. It will respond with a SYN-ACK. Because this looks like the same ACK generated in SYN-RCVD state, it will look to the client like a normal SYN/ACK packet, other than the 4WAY bit, and it will respond with a normal ACK, and the connection will complete with the normal three-way handshake.

4.3.3. Non Four-Way Server Sets 4WAY Bit

If the client decides to not continue a four-way handshake, then it will respond with an ACK and complete the normal three-way handshake. If the client decides that it does want to continue with a four-way exchange, it'll send a SYN/ACK. When the server receives the packet, the normal TCP processing will strip off the SYN, and continue processing as a normal three-way handshake.

4.4. PRE-ESTABLISHED State

When compared to the three-way handshake, the four-way handshake adds an additional RTT before the client side enters ESTABLISHED state. At first glance, this would imply that there will be an additional delay before user data can be delivered. The PRE-ESTABLISHED state addresses this issue.

From [RFC793]:

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

The PRE-ESTABLISHED state is identical to the SYN-RCVD state, except that in PRE-ESTABLISHED state we know that the connection is valid, and hence data can now be delivered to the user. It is the sending of a packet with a SYN and receiving an ACK of that SYN that provides the assurance that the connection is valid, not the transition to ESTABLISHED state. With the three-way handshake, it just happens that this corresponds exactly with entering ESTABLISHED state. With the four-way handshake, the PRE-ESTABLISHED state also corrosponds with having sent a SYN and received an ACK of that SYN, so once a connection has entered PRE-ESTABLISHED state it is also safe to deliver user data.

For the socket interface, this means that a connect() call will return upon entering either PRE-ESTABLISHED or ESTABLISHED state, and a subsequent read() on that socket will return any data that has been received. The final completion of the four-way handshake runs in parallel with delivering user data.

5. Negotiating Non-directional vs. Directional TCP Options

TCP options that are negotiated in the initial SYN exchange can be classified as either non-directional or directional. An example of a

non-directional option is the TCP Window Scale option. Negotiating a non-directional TCP option falls naturally into the Four-Way handshake, but allows for more options to be negotiated than will fit into the initial SYN packet when using expanded TCP option space. In order to allow this, the SYN/ACK from the server, with the TCP Extended Data option (EDO) [EDO], can contain initial negotiation for TCP options that weren't received in the initial SYN, which the client can then acknowledge in its SYN/ACK, using the EDO option. Because the options are non-directional, it doesn't matter which side presents it first.

Directional options do not fall as cleanly into the extended four-way handshake. A directional option is one which is originated in the initial SYN, and the servers response in the SYN/ACK is determined in direct response to the inbound option. For example, assume an option FOO that has 100 variants, where servers typically have support for all 100 variants, but clients usually only a small number. The client sends option FOO with a short list of variants that it supports, and then the server chooses which one of those to use, and responds with that variant. If instead the server initiates the the option in the SYN/ACK, it'd have to include all 100 variants and let the client choose from that list. In the future, new TCP options would need to be designed to work in the context of the four-way handshake. For existing directional options, it would not be unreasonable to require that they be included in the initial SYN, and other non-directional options would be deferred and negotiated in the SYN/ACK exchange.

5.1. Caching EDO support

One reason for using the Four-Way handshake is to allow use of the EDO option from the client, by giving the client a second chance to send TCP options, and avoid sending the EDO option in a SYN packet to a machine that doesn't understand the EDO option. An implementation could choose to cache information about peers that support the EDO option, allowing successive TCP connections to the same peer to use the EDO option in the initial SYN of a standard Three-Way handshake.

6. TCP Connection State Diagram

The following diagram is modified from the diagram in RFC 793 [RFC793]. In addition to adding the "ALLOW 4WAY?", "CONTINUE 4WAY?" and "SYN-ACK SENT" states, it also includes the three changes listed in RFC 1122 [RFC1122]:

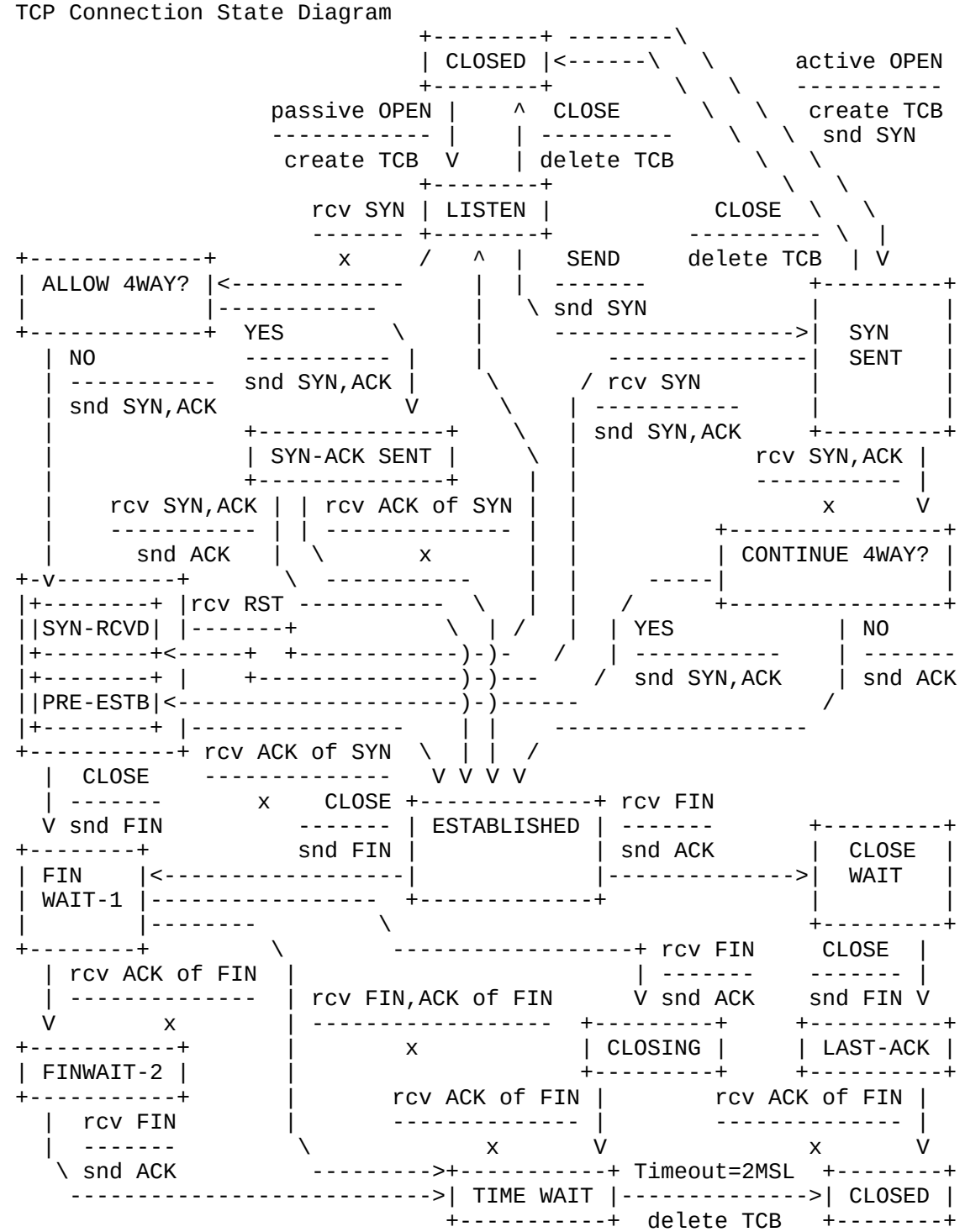
> "(a) The arrow from SYN-SENT to SYN-RCVD should be labeled with "snd SYN,ACK", to agree with the text on page 68 and with Figure 8.
>
> (b) There could be an arrow from SYN-RCVD state to LISTEN

state, conditioned on receiving a RST after a passive open (see text page 70).

(c) It is possible to go directly from FIN-WAIT-1 to the TIME-WAIT state (see page 75 of the spec)."

The SYN-RCVD and PRE-ESTABLISHED states are presented in the same box to simplify the diagram, since the transitions out of SYN-RCVD and PRE-ESTABLISHED are identical.

TCP Connection State Diagram



7. Using TCP Options in the Four-Way Handshake

7.1. New TCP Options in the SYN/ACK

The TCP protocol [RFC793] does not restrict what options may appear in which segments. The TCP protocol [RFC793] only specifies three TCP options: End of option list (EOL), No-Operation (NOP) and Maximum Segment Size (MSS), and all TCP implementations are required to support these options. "Requirements for Internet Hosts -- Communication Layers" [RFC1122] further requires that:

> A TCP MUST be able to receive a TCP option in any segment. A TCP MUST ignore without error any TCP option it does not implement, assuming that the option has a length field (all TCP options defined in the future will have length fields).

Instead, any restrictions on how an individual TCP option is to be used is specified in the definition of each individual TCP option. For example, the Maximum Segment Size option is only sent in packets with the SYN bit set [RFC793]. Prior to TCP Extensions for High Performance [RFC1072], there were no TCP options that were sent in non-SYN packets. At that time there was concern that legacy TCP implementations might not be prepared to process TCP options in non-SYN packets. To preclude that, a method of option enablement was devised: the Window Scale, Echo, and SACK Permitted options had to be exchanged in SYN packets before they could be used in non-SYN packets. (Echo and Echo Reply were replaced by Timestamps [RFC1323][RFC7323].) Additional optimization for this style of option enablement was to specify that for the Three-Way handshake, if an option wasn't received in the SYN, there was no point in putting it into the SYN/ACK since it would never be enabled. Other TCP options have different requirements.

Because option enablement that is non-directional, the Four-Way handshake allows these options to have a second chance to be enabled. The server can include additional options in its SYN/ACK that weren't present in the inbound SYN, and the client can then enable any of those additional options with its SYN/ACK. When generating the SYN/ACKs, both sides know whether or not the other side supports EDO, allowing EDO to be used to include more options than would fit in the original 40 byte option space.

Note that in the Four-Way handshake the client's SYN/ACK can omit RFC7323 style options that were in the initial SYN; if the servers SYN/ACK contained them, those options are enabled, since they were sent and received in a SYN packet. There is no way to retract an offer to enable an option, so when retransmitting a packet with a SYN, it must contain the same set of TCP options that were contained

in the original transmission of that packet, except if the option is otherwise defined. But because the clients SYN and SYN/ACK are separate packets, they can have different sets of TCP options; it is the union of the options in those two packets, intersected with the options in the servers SYN/ACK, that determine which RFC7323 style options will be enabled.

7.2. Handling Unknown TCP Options

One of the concerns since RFC 1072 [RFC1072] has been whether or not legacy TCP implementations will properly ignore unknown TCP options. As has already been stated, this has been a requirement for over 20 years, but people continue to worry about bad interactions.

Any implementation which indicates support for the Four-Way Handshake is also indicating that it properly handles unknown TCP options. This includes not only ignoring TCP options unknown to the implementation, but also properly handling known options that are received at unexpected points in the TCP stream, or that have been explicitly disabled for a specific connection.

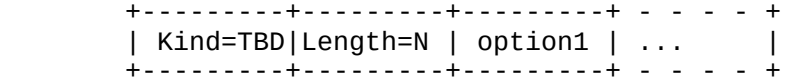
7.2.1. Unknown Option Option

One challenge with sending new options that were not enabled during the initial SYN exchange is how to decide whether or not the other side supports the option, since the other side will just silently ignore any options it doesn’t understand, so the sender has to infer non-support by the absence of any response.

The Unknown Option option is an advisory only option, that allows a receiving TCP to give explicit indication that it has received a TCP option that it does not understand.

Kind: TBD

Length: N >= 2



When a TCP receives TCP options that are not supported for this connection, in addition to silently ignoring the options [RFC1122], the TCP MAY include an Unknown Option option in the next packet it sends. It MAY generate a new ACK-only packet to send the Unknown Option option, or it MAY piggy-back it on the next packet it sends.

When a TCP receives an Unknown Option option, it SHOULD NOT send the indicated options in future packets, provided the definition

of the indicated option allows for that action. For example, when TCP-AO is being used, it is sent in every packet, so receiving an Unknown Option option indicating that TCP-AO is not supported MUST be ignored. A counter example is the UTO option [RFC5482]. If an Unknown Option option is received indicating that UTO is not supported, then the sending TCP SHOULD NOT send any more UTO options.

A TCP MUST NOT assume that a given option that it sent is supported, solely by not receiving an Unknown Option option.

7.3. TCP options

7.3.1. RFC defined options

TCP options that are defined in RFC documents include:

7.3.1.1. End of Option List

The End of Option List option [RFC793] may be present in any packet, all implementations must support EOL.

7.3.1.2. No-Operation

The No-Operation [RFC793] option may be present in any packet, all implementations must support NOP.

7.3.1.3. Maximum Segment Size

The MSS option [RFC793] only appears in SYN packets and is not negotiated, all implementations are required to support the MSS option.

7.3.1.4. Window Scale

The Window Scale option [RFC7323] uses RFC7323 style enablement, and is only sent in SYN packets. If not enabled, its contents are ignored.

7.3.1.5. SACK Permitted

The SACK Permitted option [RFC2018] uses RFC7323 style enablement, and is only sent in SYN packets. It enables use of the SACK option in non-SYN packets.

7.3.1.6. SACK

The SACK option [RFC2018] is only sent in non-SYN packets if enabled by the SACK Permitted option.

7.3.1.7. Timestamps

The Timestamps option [RFC7323] uses RFC7323 style enablement; once enabled, it is included in every packet.

7.3.1.8. MD5 Signature Option

Use of the MD5 Signature Option [RFC2385] is not negotiated. It is sent in every packet, its absence in the SYN/ACK does not disable the option, but causes the SYN/ACK to be silently dropped.

7.3.1.9. Quick-Start Response

Use of the Quick-Start Response option [RFC4782] is not negotiated. The receipt of the IP Quick-Start option implies support for the TCP Quick-Start Response option.

7.3.1.10. User Timeout Option

Though the User Timeout Option (UTO) option [RFC5482] may be exchanged in SYN packets, it is not negotiated, and may still be sent in non-SYN packets if the application has requested UTO. It relies on the fact that unknown TCP options are to be ignored [RFC1122].

7.3.1.11. TCP Authentication Options (TCP-AO)

The TCP-AO option [RFC5925] is not negotiated. The application specifies that TCP-AO is to be use and the Master Key Tuple (MKT) configuration controls when TCP-AO is to be used, and how to handle inbound packets that arrive either without TCP-AO or with an unknown MKT.

7.3.1.12. Multipath TCP (MPTCP)

The MPTCP option [RFC6824] uses a variation of RFC7323 style enablement. Each side includes the MPTCP option with its own Key for the connection in the SYN packets, if both sent and received, the final ACK contains an MPTCP option with both keys.

7.3.1.13. RFC3692-style Experiment

Two TCP options [RFC4727] are reserved for experimentation, and so whether or not they are negotiated is determined by the experiment being run.

7.3.1.14. TCP Fast Open Option

The TCP Fast Open [TFO] option spans multiple TCP connections, and is uni-directional. The first instance of it without a Cookie is used by the client to get a Cookie from the server, which is saved and can

then be used in subsequent TCP connections to allow data in the SYN-only packet to be delivered to the application before the 3WHS has been completed.

Since the Cookie information is being saved, the TCP can also save with the Cookie whether or not the other side supports the EDO option, and if so, future connections to that host can safely make use of the EDO option in the initial SYN-only packet, since it is known that the other side supports it.

In addition, the client might not include a TFO=INIT option in its first SYN-only packet due to option space limitations. But because the client has indicated support for the Four-Way handshake, the server can still safely send back in its SYN/ACK a TFO=Cookie option, allowing for TFO initialization in the client.

7.3.1.15. T/TCP Options

T/TCP [RFC1644] defines three TCP options: CC (connection count), CC.NEW and CC.ECHO. Though now obsolete [RFC6247], this is a good example of a directional TCP option. The CC or CC.NEW option is sent in the initial SYN, and the responding SYN/ACK has a CC.ECHO option that contains the connection count from the received CC or CC.NEW option. This is not an RFC7323 style enablement, and only fits into the Four-Way handshake if the CC or CC.NEW option is included in the initial SYN packet, the servers SYN/ACK can’t contain a CC.ECHO if it didn’t receive a CC option. However, T/TCP is aimed at reducing the Three-Way handshake to a two packet exchange, and needs to keep state about which hosts can utilize T/TCP. As such, the client could keep state as to which servers support the EDO option, and then be able make use of the EDO option in its initial SYN for future connections to those servers.

8. IANA Considerations

A Kind value needs to be allocated for the Unknown Option Option.

9. Security Considerations

TBD

10. References

10.1. Normative References

[RFC793] Postel, J., "Transmission Control Protocol - DARPA Internet Program Protocol Specification", RFC 793, DARPA, September 1981.

[RFC1122] Braden, R., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, October 1989, <http://www.rfc-editor.org/info/rfc1122>.

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

10.2. Informative References

[Borman14] Borman, D., "Re: [tcpm] New Version Notification for draft-touch-tcpm-tcp-edo-01.txt", message to the TCPM mailing list, 22 May 2014, <http://www.ietf.org/mail-archive/web/tcpm/current/msg08804.html>.

[RFC1072] Jacobson, V., and R.T. Braden, "TCP extensions for long-delay paths", RFC 1072, October 1988, <http://www.rfc-editor.org/info/rfc1072>.

[RFC1323] Jacobson, V., Braden, R., and D. Borman. "TCP Extensions for High Performance." RFC 1323, May 1992, <http://www.rfc-editor.org/info/rfc1323>.

[RFC1644] R. Braden, "T/TCP -- TCP Extensions for Transactions Functional Specification", RFC 1644, July 1994, <http://www.rfc-editor.org/info/rfc1644>.

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

[RFC2385] A. Heffernan, "Protection of BGP Sessions via the TCP MD5 Signature Option", RFC 2385, August 1998, <http://www.rfc-editor.org/info/rfc2385>.

[RFC4727] B. Fenner, "Experimental Values In IPv4, IPv6, ICMPv4, ICMPv6, UDP, and TCP Headers", RFC 4727, November 2006, <http://www.rfc-editor.org/info/rfc4727>.

[RFC4782] Floyd, S., Allman, M., Jain, A., and P. Sarolahti, "Quick-Start for TCP and IP", RFC 4782, January 2007, <http://www.rfc-editor.org/info/rfc4782>.

[RFC5482] Eggert, L., and F. Gont, "TCP User Timeout Option", RFC 5482, March 2009, <http://www.rfc-editor.org/info/rfc5482>.

[RFC5925] Touch, J., Mankin, A., and R. Bonica, "The TCP Authentication Option", RFC 5925, June 2010, <http://www.rfc-editor.org/info/rfc5925>.

[RFC6247] L. Eggert, "Moving the Undeployed TCP Extensions RFC 1072, RFC 1106, RFC 1110, RFC 1145, RFC 1146, RFC 1379, RFC 1644, and RFC 1693 to Historic Status", RFC 6247, May 2011, <http://www.rfc-editor.org/info/rfc6247>.

[RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, January 2013, <http://www.rfc-editor.org/info/rfc6824>.

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

[RFC7323] Borman, D., Braden, R., Jacobson, V., and R. Scheffenegger, Ed., "TCP Extension for High Performance", RFC 7323, September 2014, <http://www.rfc-editor.org/info/rfc7323>.

[TFO] Cheng, Y., Jhu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, December 2014, <http://www.rfc-editor.org/info/rfc7413>.

[EDO] Joe Touch, J., and W. Eddy, "TCP Extended Data Offset Option", Work in Progress, draft-ietf-tcpm-tcp-edo-01.txt, October 2014.

[Kohler04] Kohler, E, "Extended Option Space for TCP" Work in Progress, draft-kohler-tcpm-extopt-00.txt, September 2004.

[Touch14] Touch, J., Briscoe, B., and T. Faber, "TCP SYN Extended Option Space in the Payload of a Supplementary Segment", Work in Progress, draft-touch-tcpm-tcp-syn-ext-opt-01.txt, September 2014.

[Eddy08] Eddy, W., and A. Langley, "Extending the Space Available for TCP Options", Work in Progress, draft-eddy-tcp-loo-04, July 2008.

[Yourtchenko11] Yourtchenko, A., "Introducing TCP Long Options by Invalid Checksum", Work in Progress, draft-yourtchenko-tcp-loic-00.txt, April 2011.

[Ramaiah12] Ramaiah, A., "TCP option space extension", Work in Progress, draft-ananth-tcpm-tcpoptext-00.txt, March 2012.

[Briscoe14] Briscoe, B., "Extended TCP Option Space in the Payload of an Alternative SYN", Work in Progress, draft-briscoe-tcpm-syn-op-sis-02, September 2014.

Appendix A. First Response of the Four-Way Handshake

For a connection with ISS values of ISSA from the client and ISSB from the server, three different options for the first server response were considered:

(1) SYN(seq=ISSB)
(2) SYN(seq=ISSB)/ACK(seq=ISSA-1)
(3) SYN(Seq=ISSB)/ACK(seq=ISSA)

SYN(seq=ISSB)

The original idea for the four-way handshake was to have the server do a simple turn-around of the TCP three-way handshake, by responding to the initial SYN with another bare SYN. Because it had already received a SYN and knows that the client supports the four-way handshake, it could respond with a plain SYN, making use of header modifying options that the client had indicated it supported. This is similar to a a simultaneous open, except the server is able to transition from SYN-SENT to ESTABLISHED instead of going through SYN-RECEIVED state.

```
    Enter SYN-SENT
    SYN(seq=ISSA) ->
                                        Enter SYN-SENT
                                    <- SYN(seq=ISSB)
    Enter SYN-RECEIVED
    SYN(seq=ISSA)/ACK(ISSB) ->
                                        Enter ESTABLISHED
                                    <- ACK(ISSA)
    Enter ESTABLISHED
```

The problems with this approach are that it forces the full four-way handshake, and a middle-box in the path might block the returning bare SYN.

SYN(seq=ISSB)/ACK(seq=ISSA-1)

This response also turns the three-way handshake into something that looks a lot like a simultaneous open, since the ACK does not acknowledge the SYN. The disadvantage is that it also forces a full four-way handshake, since it does not acknowledge the initial SYN. However, this should work better for getting through a middle-box since it is not a bare SYN. But if the middle-box is digging into the TCP packet and tries to verify the ACK field, it might still block this packet since it is not the expected ACK field of the normal three-way handshake.

SYN(seq=ISSB)/ACK(seq=ISSA)

This response looks like the normal three-way handshake response, which gives the client the ability to choose whether to complete the three-way handshake by sending an ACK(ISSB), or continue the four-way handshake by responding with SYN(seq=ISSA)/ACK(ISSB). The advantage of this option is that it doesn't always force the four-way handshake, and to a middle-box the packets look like the normal TCP packets that it expects to see.

The third option offers the least possibility that middle-boxes will block the packets, and also leaves the flexibility for deciding on a three-way or four-way handshake up to the client. Because it is to the client's benefit to have a four-way handshake, it should be the one to decide whether or not the four-way handshake is needed for a particular handshake.

Appendix B. Communicating Four-Way Handshake Support

Besides allocating a 4WAY bit in the TCP header, two other options were considered for communicating support for the four-way handshake:

Create a new 4WAY TCP option

This does not have the interoperability issues that the 4WAY TCP bit has, because it is assumed that connections will not send unknown TCP options. The disadvantage of this is that it requires two more bytes out of the TCP option space.

Implied support by other TCP options

The primary motivation for the four-way handshake is to give the client a second chance to send TCP options in a SYN. This is intended for use with the new TCP EDO option, and the presence of the EDO option could imply support for the four-way handshake. This allows the client to send additional TCP options using the TCP EDO option in a SYN/ACK packet.

Acknowledgments

TBD

Contributors

TBD

Author's Address

David Borman
Quantum Corporation
1155 Centre Pointe Drive, Suite 1
Mendota Heights, MN 55120

Phone: (651) 688-4394
Email: david.borman@quantum.com

Internet Engineering Task Force
Internet-Draft
Obsoletes: 793, 879, 6093, 6528, 6691
(if approved)
Updates: 1122 (if approved)
Intended status: Standards Track
Expires: August 10, 2015

W. Eddy, Ed.
MTI Systems

February 6, 2015

Transmission Control Protocol Specification
draft-eddy-rfc793bis-05

Abstract

This document specifies the Internet's Transmission Control Protocol (TCP). TCP is an important transport layer protocol in the Internet stack, and has continuously evolved over decades of use and growth of the Internet. Over this time, a number of changes have been made to TCP as it was specified in RFC 793, though these have only been documented in a piecemeal fashion. This document collects and brings those changes together with the protocol specification from RFC 793. This document obsoletes RFC 793 and several other RFCs (TODO: list all actual RFCs when finished).

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

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [1].

Status of This Memo

This Internet-Draft will expire on August 10, 2015.

Table of Contents

1. Purpose and Scope 3
2. Introduction 4
3. Functional Specification 4
3.1. Header Format 4
3.2. Terminology 9
3.3. Sequence Numbers 13
3.4. Establishing a connection 20
3.5. Closing a Connection 27
3.6. Precedence and Security 29
3.7. Segmentation 30
3.7.1. Maximum Segment Size Option 31
3.7.2. Path MTU Discovery 32
3.7.3. Interfaces with Variable MSS Values 32
3.7.4. IPv6 Jumbograms 32
3.8. Data Communication 32
3.9. Interfaces 36
3.9.1. User/TCP Interface 37

3.9.2. TCP/Lower-Level Interface 43
3.10. Event Processing 44
3.11. Glossary 67
4. Changes from RFC 793 72
5. IANA Considerations 75
6. Security and Privacy Considerations 75
7. Acknowledgements 76
8. References 76
8.1. Normative References 76
8.2. Informative References 76
Appendix A. TCP Requirement Summary 77
Author's Address 80

1. Purpose and Scope

In 1981, RFC 793 [2] was released, documenting the Transmission Control Protocol (TCP), and replacing earlier specifications for TCP that had been published in the past.

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

For several decades, RFC 793 plus a number of other documents have combined to serve as the specification for TCP [10]. Over time, a number of errata have been identified on RFC 793, as well as deficiencies in security, performance, and other aspects. A number of enhancements has grown and been documented separately. These were never accumulated together into an update to the base specification.

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

In addition to the protocol specification that descibes the TCP segment format, generation, and processing rules that are to be implemented in code, RFC 793 and other updates also contain informative and descriptive text for human readers to understand aspects of the protocol design and operation. This document does not

attempt to alter or update this informative text, and is focused only on updating the normative protocol specification. We preserve references to the documentation containing the important explanations and rationale, where appropriate.

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

2. Introduction

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

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

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

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

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

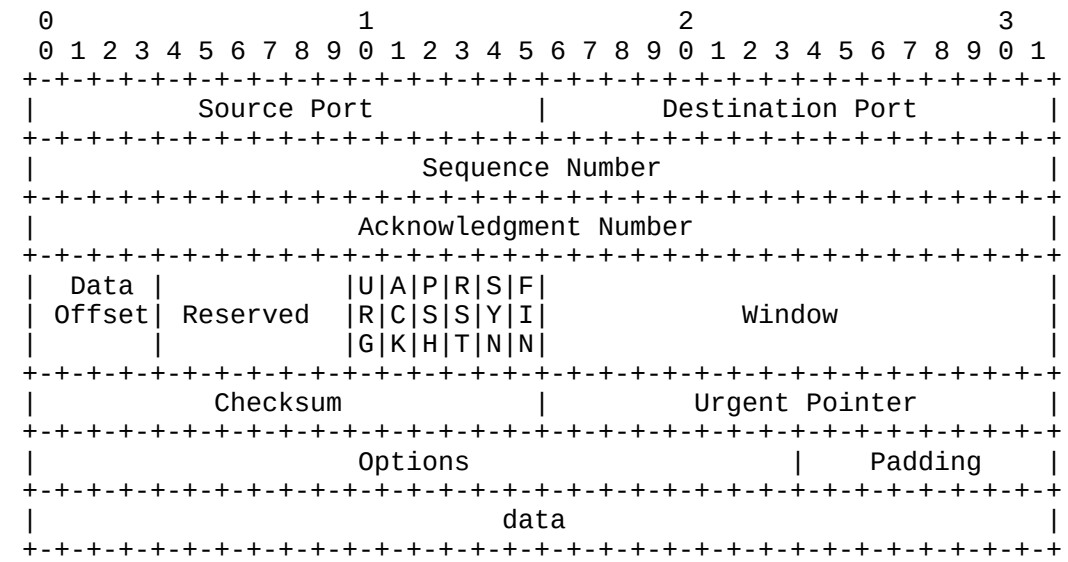
TEMPORARY EDITOR'S NOTE: the current revision of this document does not yet collect all of the changes that will be in the final version. The set of content changes planned for future revisions is kept in Section 4.

3. Functional Specification

3.1. Header Format

TCP segments are sent as internet datagrams. The Internet Protocol header carries several information fields, including the source and destination host addresses [2]. A TCP header follows the internet header, supplying information specific to the TCP protocol. This division allows for the existence of host level protocols other than TCP.

TCP Header Format



TCP Header Format

Note that one tick mark represents one bit position.

Figure 1

Source Port: 16 bits

The source port number.

Destination Port: 16 bits

The destination port number.

Sequence Number: 32 bits

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

Acknowledgment Number: 32 bits

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

Data Offset: 4 bits

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

Reserved: 6 bits

Reserved for future use. Must be zero.

Control Bits: 6 bits (from left to right):

URG: Urgent Pointer field significant
ACK: Acknowledgment field significant
PSH: Push Function
RST: Reset the connection
SYN: Synchronize sequence numbers
FIN: No more data from sender

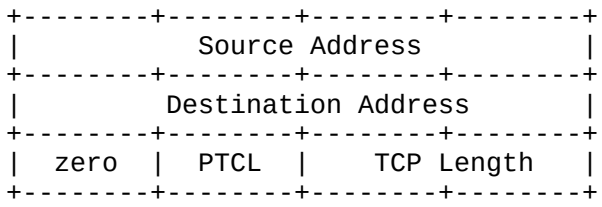
Window: 16 bits

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

Checksum: 16 bits

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

The checksum also covers a 96 bit pseudo header conceptually prefixed to the TCP header. This pseudo header contains the Source Address, the Destination Address, the Protocol, and TCP length. This gives the TCP protection against misrouted segments. This information is carried in the Internet Protocol and is transferred across the TCP/Network interface in the arguments or results of calls by the TCP on the IP.



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

Urgent Pointer: 16 bits

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

Options: variable

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

Case 1: A single octet of option-kind.

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

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

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

Currently defined options include (kind indicated in octal):

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

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

Specific Option Definitions

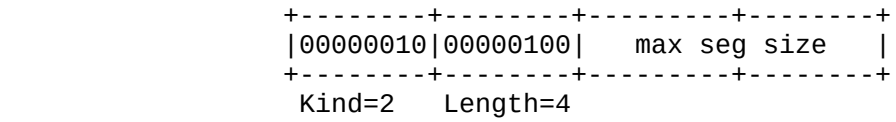
End of Option List

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

No-Operation

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

Maximum Segment Size (MSS)



Maximum Segment Size Option Data: 16 bits

If this option is present, then it communicates the maximum receive segment size at the TCP which sends this segment. This

field may be sent in the initial connection request (i.e., in segments with the SYN control bit set) and must not be sent in other segments. If this option is not used, any segment size is allowed.

Padding: variable

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

3.2. Terminology

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

Send Sequence Variables

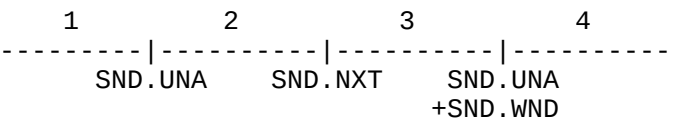
```
SND.UNA - send unacknowledged
SND.NXT - send next
SND.WND - send window
SND.UP  - send urgent pointer
SND.WL1 - segment sequence number used for last window update
SND.WL2 - segment acknowledgment number used for last window
          update
ISS     - initial send sequence number
```

Receive Sequence Variables

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

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

Send Sequence Space



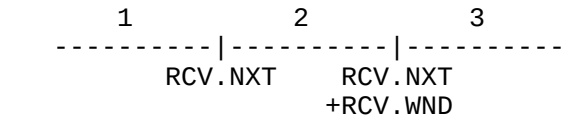
1 - old sequence numbers which have been acknowledged
2 - sequence numbers of unacknowledged data
3 - sequence numbers allowed for new data transmission
4 - future sequence numbers which are not yet allowed

Send Sequence Space

Figure 2

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

Receive Sequence Space



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

Receive Sequence Space

Figure 3

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

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

Current Segment Variables

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

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

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

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

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

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

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

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

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

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

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

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

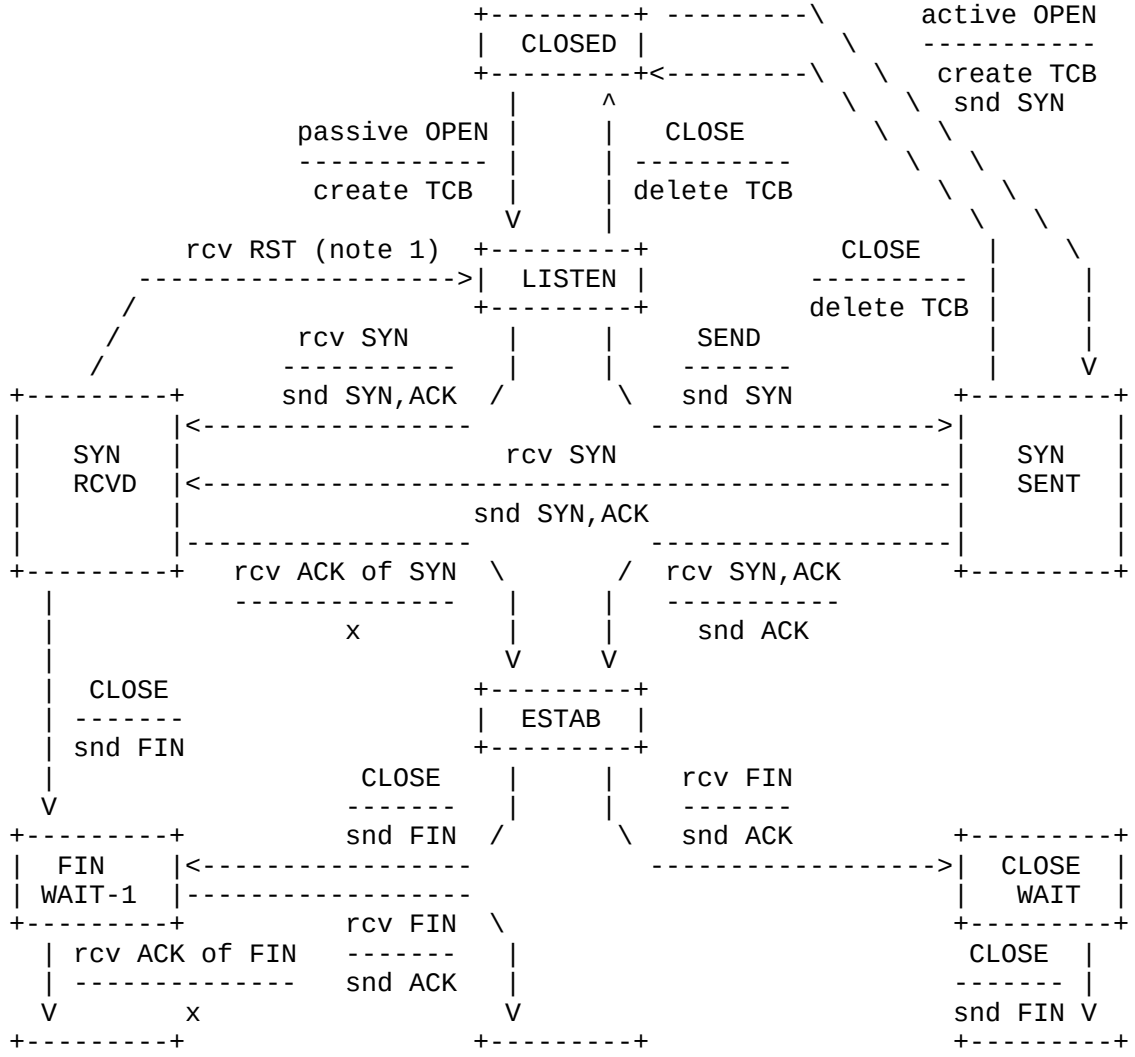
CLOSED - represents no connection state at all.

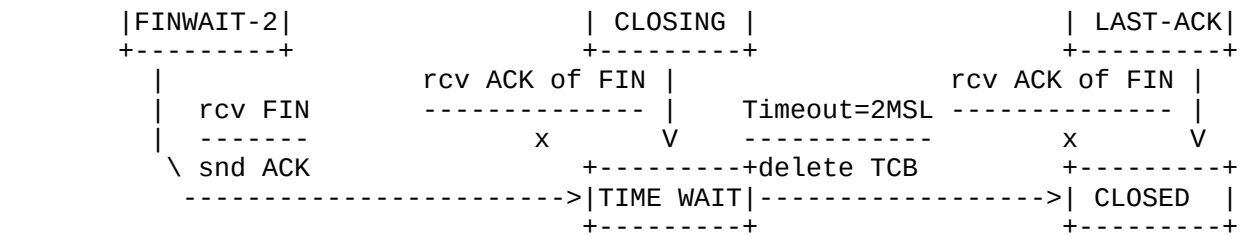
A TCP connection progresses from one state to another in response to events. The events are the user calls, OPEN, SEND, RECEIVE, CLOSE,

ABORT, and STATUS; the incoming segments, particularly those containing the SYN, ACK, RST and FIN flags; and timeouts.

The state diagram in Figure 4 illustrates only state changes, together with the causing events and resulting actions, but addresses neither error conditions nor actions which are not connected with state changes. In a later section, more detail is offered with respect to the reaction of the TCP to events.

NOTA BENE: this diagram is only a summary and must not be taken as the total specification.





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

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

TCP Connection State Diagram

Figure 4

3.3. Sequence Numbers

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

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

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

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

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

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

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

SND.UNA = oldest unacknowledged sequence number

SND.NXT = next sequence number to be sent

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

SEG.SEQ = first sequence number of a segment

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

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

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

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

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

When data is received the following comparisons are needed:

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

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

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

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

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

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

or

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

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

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

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

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

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

occurs, while the FIN is considered to occur after the last actual data octet in a segment in which it occurs. The segment length (SEG.LEN) includes both data and sequence space occupying controls. When a SYN is present then SEG.SEQ is the sequence number of the SYN.

Initial Sequence Number Selection

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

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

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

TCP 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"). F() MUST NOT be computable from the outside, or an attacker could still guess at sequence numbers from the ISN used for some other connection. The PRF could be implemented as a cryptographic has of the concatenation of the TCP connection parameters and some secret data. For discussion of the selection of a specific hash algorithm and management of the secret key data, please see Section 3 of [8].

For each connection there is a send sequence number and a receive sequence number. The initial send sequence number (ISS) is chosen by

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

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

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

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

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

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

Knowing When to Keep Quiet

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

The TCP Quiet Time Concept

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

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

Under normal conditions, TCPs keep track of the next sequence number to emit and the oldest awaiting acknowledgment so as to avoid mistakenly using a sequence number over before its first use has been acknowledged. This alone does not guarantee that old duplicate data is drained from the net, so the sequence space has been made very large to reduce the probability that a wandering duplicate will cause trouble upon arrival. At 2 megabits/sec. it takes 4.5 hours to use up 2**32 octets of sequence space. Since the maximum segment lifetime in the net is not likely to exceed a few tens of seconds, this is deemed ample protection for foreseeable nets, even if data rates escalate to l0’s of megabits/sec. At 100 megabits/sec, the cycle time is 5.4 minutes which may be a little short, but still within reason.

The basic duplicate detection and sequencing algorithm in TCP can be defeated, however, if a source TCP does not have any memory of the sequence numbers it last used on a given connection. For example, if the TCP were to start all connections with sequence number 0, then upon crashing and restarting, a TCP might re-form an earlier connection (possibly after half-open connection resolution) and emit packets with sequence numbers identical to or overlapping with

packets still in the network which were emitted on an earlier incarnation of the same connection. In the absence of knowledge about the sequence numbers used on a particular connection, the TCP specification recommends that the source delay for MSL seconds before emitting segments on the connection, to allow time for segments from the earlier connection incarnation to drain from the system.

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

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

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

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

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

potential sequence number overlap area which could cause confusion at the receiver.

3.4. Establishing a connection

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

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

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

```
      TCP A                                                TCP B

1.  CLOSED                                               LISTEN

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

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

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

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

Basic 3-Way Handshake for Connection Synchronization

Figure 5

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

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

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

```
      TCP A                                                TCP B

1.  CLOSED                                               CLOSED

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

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

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

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

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

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

Simultaneous Connection Synchronization

Figure 6

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

```
    TCP A                                                TCP B

1.  CLOSED                                               LISTEN

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

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

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

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

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

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

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

Recovery from Old Duplicate SYN

Figure 7

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

Half-Open Connections and Other Anomalies

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

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

site B TCP that something is wrong, and it is expected to abort the connection.

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

```
      TCP A                                              TCP B

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

  2.  CLOSED                                          ESTABLISHED

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

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

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

  6.  SYN-SENT                                         CLOSED

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

Half-Open Connection Discovery

Figure 8

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

An interesting alternative case occurs when TCP A crashes and TCP B tries to send data on what it thinks is a synchronized connection.

This is illustrated in Figure 9. In this case, the data arriving at TCP A from TCP B (line 2) is unacceptable because no such connection exists, so TCP A sends a RST. The RST is acceptable so TCP B processes it and aborts the connection.

```
       TCP A                                                TCP B

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

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

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

Active Side Causes Half-Open Connection Discovery

Figure 9

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

```
    TCP A                                            TCP B

1.  LISTEN                                           LISTEN

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

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

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

5.  LISTEN                                           LISTEN
```

Old Duplicate SYN Initiates a Reset on two Passive Sockets

Figure 10

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

Reset Generation

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

There are three groups of states:

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

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

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

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

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

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

and an acknowledgment indicating the next sequence number expected to be received, and the connection remains in the same state.

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

Reset Processing

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

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

3.5. Closing a Connection

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

There are essentially three cases:

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

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

3) Both users CLOSE simultaneously

Case 1: Local user initiates the close

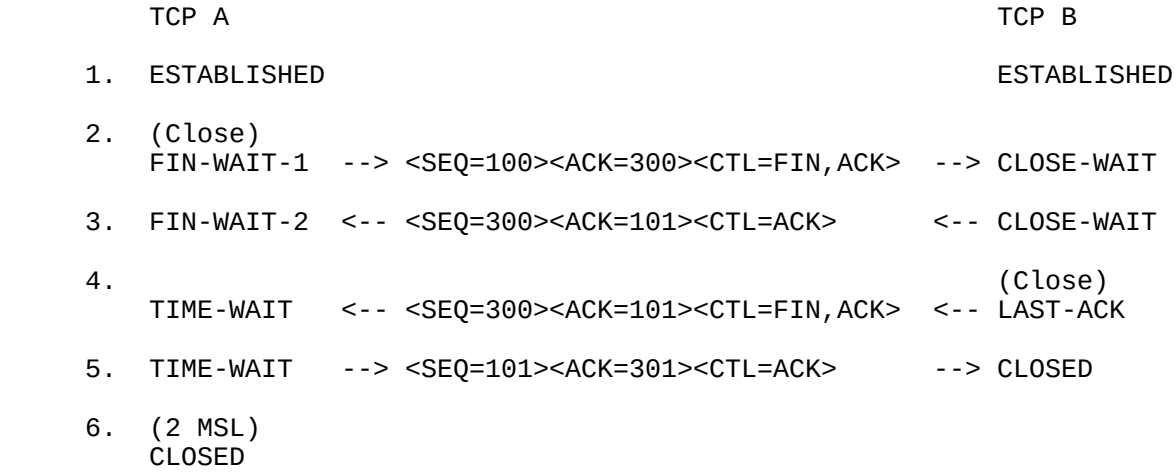
In this case, a FIN segment can be constructed and placed on the outgoing segment queue. No further SENDs from the user will be accepted by the TCP, and it enters the FIN-WAIT-1 state. RECEIVEs are allowed in this state. All segments preceding and including FIN will be retransmitted until acknowledged.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

Case 2: TCP receives a FIN from the network

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

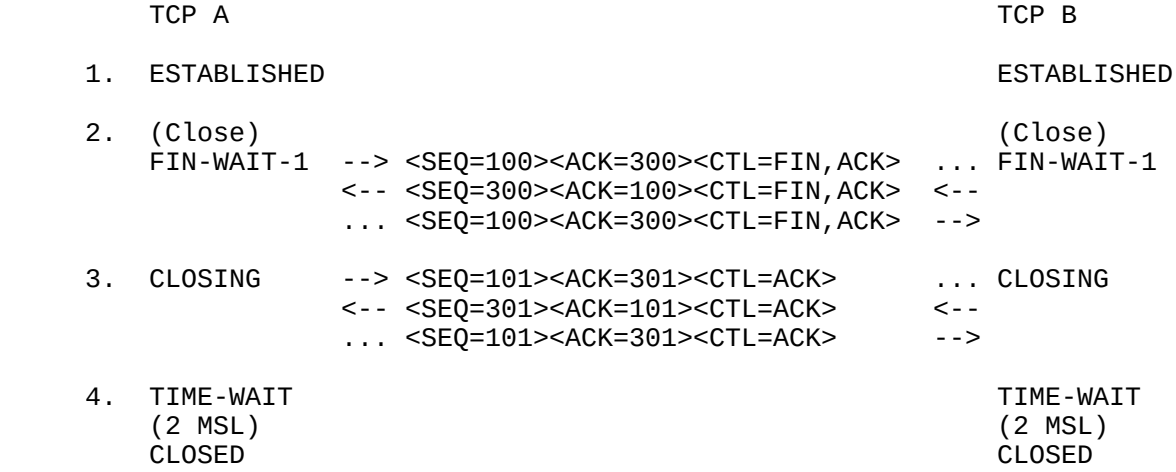
Case 3: both users close simultaneously

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



Normal Close Sequence

Figure 11



Simultaneous Close Sequence

Figure 12

3.6. Precedence and Security

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

The precedence and security parameters used in TCP are exactly those defined in the Internet Protocol (IP) [2]. Throughout this TCP specification the term "security/compartment" is intended to indicate the security parameters used in IP including security, compartment, user group, and handling restriction.

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

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

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

3.7. Segmentation

The term "segmentation" refers to the activity TCP performs when ingesting a stream of bytes from a sending application and packetizing that stream of bytes into TCP segments.

For efficiency and performance reasons, it is desirable to send large segments that contain as many bytes of payload data as possible. However, packets that are too long will either be fragmented or dropped within the network. Some firewalls or middleboxes may drop fragmented packets. In either case, when packets are dropped, the connection can fail; hence, it is best for a TCP implementation to avoid generating fragments.

To enable a TCP sender to maximize the size of segments that it sends, without generating fragments, TCP includes the Maximum Segment Size option to convey endpoint information, and TCP implementations also support Path MTU Discovery to discover the limits and capabilites of intermediate networks.

When TCP is used in a situation where either the IP or TCP headers are not minimum, 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. The rationale for this is explained in RFC 6691.

3.7.1. Maximum Segment Size Option

TCP MUST implement both sending and receiving the Maximum Segment Size option.

TCP SHOULD send an MSS (Maximum Segment Size) option in every SYN segment when its receive MSS differs from the default 536, and MAY send it always.

If an MSS option is not received at connection setup, TCP MUST assume a default send MSS of 536 (576-40).

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

```
    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 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; this is normally 20, but may be larger if TCP options are to be sent.
- IPoptionsize is the size of any IP options that TCP will pass to the IP layer with the current message.

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 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). TCP obtains MMS_R and MMS_S from the IP layer; see the generic call GET_MAXSIZES in Section 3.4 of RFC 1122.

3.7.2. Path MTU Discovery

The TCP MSS option specifies an upper bound for the size of packets that can be received. Hence, setting the value in the MSS option too small can impact the ability for Path MTU Discovery to find a larger path MTU. For more information on Path MTU Discovery, see:

- "Path MTU Discovery" [RFC1191]
- "TCP Problems with Path MTU Discovery" [RFC2923]
- "Packetization Layer Path MTU Discovery" [RFC4821]

3.7.3. Interfaces with Variable MSS Values

The effective MTU can sometimes vary, as when used with variable compression, e.g., RObust Header Compression (ROHC) [RFC5795]. It is tempting for TCP to want to advertise the largest possible MSS, to support the most efficient use of compressed payloads. Unfortunately, some compression schemes occasionally need to transmit full headers (and thus smaller payloads) to resynchronize state at their endpoint compressors/decompressors. If the largest MTU is used to calculate the value to advertise in the MSS option, TCP retransmission may interfere with compressor resynchronization.

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

3.7.4. IPv6 Jumbograms

In order to support TCP over IPv6 jumbograms, implementations need to be able to send TCP segments larger than 64K. RFC 2675 [RFC2675] defines that a value of 65,535 is to be treated as infinity, and Path MTU Discovery [RFC1981] is used to determine the actual MSS.

3.8. Data Communication

Once the connection is established data is communicated by the exchange of segments. Because segments may be lost due to errors (checksum test failure), or network congestion, TCP uses retransmission (after a timeout) to ensure delivery of every segment. Duplicate segments may arrive due to network or TCP retransmission.

As discussed in the section on sequence numbers the TCP performs certain tests on the sequence and acknowledgment numbers in the segments to verify their acceptability.

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

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

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

Retransmission Timeout

NOTE: TODO this needs to be updated in light of 1122 4.2.2.15 and errata 573; this will be done as part of RFC 1122 incorporation into this document.
Because of the variability of the networks that compose an internetwork system and the wide range of uses of TCP connections the retransmission timeout must be dynamically determined. One procedure for determining a retransmission timeout is given here as an illustration.

An Example Retransmission Timeout Procedure

Measure the elapsed time between sending a data octet with a particular sequence number and receiving an acknowledgment that covers that sequence number (segments sent do not have to match segments received). This measured elapsed time is the Round Trip Time (RTT). Next compute a Smoothed Round Trip Time (SRTT) as:

SRTT = (ALPHA * SRTT) + ((1-ALPHA) * RTT)

and based on this, compute the retransmission timeout (RTO) as:

RTO = min[UBOUND,max[LBOUND,(BETA*SRTT)]]

where UBOUND is an upper bound on the timeout (e.g., 1 minute), LBOUND is a lower bound on the timeout (e.g., 1 second), ALPHA is a smoothing factor (e.g., .8 to .9), and BETA is a delay variance factor (e.g., 1.3 to 2.0).

The Communication of Urgent Information

As a result of implementation differences and middlebox interactions, new applications SHOULD NOT employ the TCP urgent mechanism. However, TCP implementations MUST still include support for the urgent mechanism. Details can be found in RFC 6093 [7].

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

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

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

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

A TCP MUST support a sequence of urgent data of any length. [3]

A TCP MUST inform the application layer asynchronously whenever it receives an Urgent pointer and there was previously no pending urgent data, or whenvever the Urgent pointer advances in the data stream. There MUST be a way for the application to learn how much urgent data remains to be read from the connection, or at least to determine whether or not more urgent data remains to be read. [3]

Managing the Window

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

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

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

The sending TCP must be prepared to accept from the user and send at least one octet of new data even if the send window is zero. The sending TCP must regularly retransmit to the receiving TCP even when the window is zero. Two minutes is recommended for the retransmission interval when the window is zero. This retransmission is essential to guarantee that when either TCP has a zero window the re-opening of the window will be reliably reported to the other.

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

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

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

The window management procedure has significant influence on the communication performance. The following comments are suggestions to implementers.

Window Management Suggestions

Allocating a very small window causes data to be transmitted in many small segments when better performance is achieved using fewer large segments.

One suggestion for avoiding small windows is for the receiver to defer updating a window until the additional allocation is at least X percent of the maximum allocation possible for the connection (where X might be 20 to 40).

Another suggestion is for the sender to avoid sending small segments by waiting until the window is large enough before sending data. If the user signals a push function then the data must be sent even if it is a small segment.

Note that the acknowledgments should not be delayed or unnecessary retransmissions will result. One strategy would be to send an acknowledgment when a small segment arrives (with out updating the window information), and then to send another acknowledgment with new window information when the window is larger.

The segment sent to probe a zero window may also begin a break up of transmitted data into smaller and smaller segments. If a segment containing a single data octet sent to probe a zero window is accepted, it consumes one octet of the window now available. If the sending TCP simply sends as much as it can whenever the window is non zero, the transmitted data will be broken into alternating big and small segments. As time goes on, occasional pauses in the receiver making window allocation available will result in breaking the big segments into a small and not quite so big pair. And after a while the data transmission will be in mostly small segments.

The suggestion here is that the TCP implementations need to actively attempt to combine small window allocations into larger windows, since the mechanisms for managing the window tend to lead to many small windows in the simplest minded implementations.

3.9. Interfaces

There are of course two interfaces of concern: the user/TCP interface and the TCP/lower-level interface. We have a fairly elaborate model of the user/TCP interface, but the interface to the lower level

protocol module is left unspecified here, since it will be specified in detail by the specification of the lower level protocol. For the case that the lower level is IP we note some of the parameter values that TCPs might use.

3.9.1. User/TCP Interface

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

TCP User Commands

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

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

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

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

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

Open

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

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

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

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

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

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

The TCP or some component of the operating system will verify the users authority to open a connection with the specified precedence or security/compartment. The absence of precedence 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 and only if the precedence is equal to or higher than the precedence requested in the OPEN call.

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

or that any attempt to raise the precedence be confirmed by the user.

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

Send

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

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

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

New applications SHOULD NOT set the URGENT flag [7] due to implementation differences and middlebox issues.

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

If no foreign socket was specified in the OPEN, but the connection is established (e.g., because a LISTENing connection has become specific due to a foreign segment arriving for the local socket), then the designated buffer is sent to the implied foreign socket. Users who make use of OPEN with an

unspecified foreign socket can make use of SEND without ever explicitly knowing the foreign socket address.

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

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

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

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

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

Receive

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

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

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

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

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

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

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

Close

Format: CLOSE (local connection name)

This command causes the connection specified to be closed. If the connection is not open or the calling process is not authorized to use this connection, an error is returned. Closing connections is intended to be a graceful operation in

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

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

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

Close also implies push function.

Status

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

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

This command returns a data block containing the following information:

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

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

Abort

Format: ABORT (local connection name)

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

TCP-to-User Messages

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

The following information is provided:

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

3.9.2. TCP/Lower-Level Interface

The TCP calls on a lower level protocol module to actually send and receive information over a network. One case is that of the ARPA internetwork system where the lower level module is the Internet Protocol (IP) [2].

If the lower level protocol is IP it provides arguments for a type of service and for a time to live. TCP uses the following settings for these parameters:

Type of Service = Precedence: given by user, Delay: normal, Throughput: normal, Reliability: normal; or binary XXX00000, where XXX are the three bits determining precedence, e.g. 000 means routine precedence.

Time to Live = one minute, or 00111100.

Note that the assumed maximum segment lifetime is two minutes. Here we explicitly ask that a segment be destroyed if it cannot be delivered by the internet system within one minute.

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.

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.

3.10. Event Processing

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

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

Events that occur:

User Calls

OPEN
SEND
RECEIVE
CLOSE
ABORT
STATUS

Arriving Segments

```
      SEGMENT ARRIVES

   Timeouts

      USER TIMEOUT
      RETRANSMISSION TIMEOUT
      TIME-WAIT TIMEOUT
```

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

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

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

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

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

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

OPEN Call

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

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

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

LISTEN STATE

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

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

         Return "error: connection already exists".
```

SEND Call

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

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

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

LISTEN STATE

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

SYN-SENT STATE
SYN-RECEIVED STATE

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

ESTABLISHED STATE
CLOSE-WAIT STATE

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

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

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

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

RECEIVE Call

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

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

Otherwise return "error: connection does not exist".

LISTEN STATE
SYN-SENT STATE
SYN-RECEIVED STATE

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

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

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

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

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

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

CLOSE-WAIT STATE

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

CLOSING STATE
LAST-ACK STATE

```
      TIME-WAIT STATE

         Return "error: connection closing".
```

CLOSE Call

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

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

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

LISTEN STATE

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

SYN-SENT STATE

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

SYN-RECEIVED STATE

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

ESTABLISHED STATE

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

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

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

CLOSE-WAIT STATE

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

CLOSING STATE
LAST-ACK STATE
TIME-WAIT STATE

Respond with "error: connection closing".

ABORT Call

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

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

Otherwise return "error: connection does not exist".

LISTEN STATE

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

SYN-SENT STATE

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

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

Send a reset segment:

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

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

CLOSING STATE LAST-ACK STATE TIME-WAIT STATE

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

STATUS Call

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

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

Otherwise return "error: connection does not exist".

LISTEN STATE

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

SYN-SENT STATE

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

SYN-RECEIVED STATE

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

ESTABLISHED STATE

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

FIN-WAIT-1 STATE

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

FIN-WAIT-2 STATE

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

CLOSE-WAIT STATE

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

CLOSING STATE

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

LAST-ACK STATE

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

TIME-WAIT STATE

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

SEGMENT ARRIVES

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

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

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

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

If the ACK bit is on,

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

Return.

If the state is LISTEN then

first check for an RST

An incoming RST should be ignored. Return.

second check for an ACK

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

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

Return.

third check for a SYN

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

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

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

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

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

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

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

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

fourth other text or control

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

If the state is SYN-SENT then

first check the ACK bit

If the ACK bit is set

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

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

and discard the segment. Return.

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

second check the RST bit

If the RST bit is set

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

third check the security and precedence

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

If there is an ACK

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

Otherwise

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

If there is an ACK

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

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

If there is no ACK

If the precedence in the segment is higher than the precedence in the TCB then if allowed by the user and the system raise the precedence in the TCB to that in the segment, if not allowed to raise the prec then send a reset.

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

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

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

fourth check the SYN bit

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

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

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

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

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

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

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

and send it. Set the variables:

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

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

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

Otherwise,

first check sequence number

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

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

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

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

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

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

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

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

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

In the following it is assumed that the segment is the idealized segment that begins at RCV.NXT and does not exceed the window. One could tailor actual segments to fit this assumption by trimming off any portions that lie outside the window (including SYN and FIN), and only processing further

if the segment then begins at RCV.NXT. Segments with higher beginning sequence numbers should be held for later processing.

second check the RST bit,

SYN-RECEIVED STATE

If the RST bit is set

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

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

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

CLOSING STATE
LAST-ACK STATE
TIME-WAIT

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

third check security and precedence

SYN-RECEIVED

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

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

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

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

fourth, check the SYN bit,

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

TODO: need to incorporate RFC 1122 4.2.2.20(e) here

If the SYN is in the window it is an error, send a reset, any outstanding RECEIVEs and SEND should receive "reset" responses, all segment queues should be flushed, the user should also receive an unsolicited general "connection reset" signal, enter the CLOSED state, delete the TCB, and return.

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

fifth check the ACK field,

if the ACK bit is off drop the segment and return

if the ACK bit is on

SYN-RECEIVED STATE

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

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

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

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

and send it.

ESTABLISHED STATE

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

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

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

FIN-WAIT-1 STATE

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

FIN-WAIT-2 STATE

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

CLOSE-WAIT STATE

Do the same processing as for the ESTABLISHED state.

CLOSING STATE

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

LAST-ACK STATE

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

TIME-WAIT STATE

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

sixth, check the URG bit,

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

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

CLOSE-WAIT STATE

CLOSING STATE
LAST-ACK STATE
TIME-WAIT

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

seventh, process the segment text,

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

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

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

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

Please note the window management suggestions in section 3.7.

Send an acknowledgment of the form:

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

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

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

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

eighth, check the FIN bit,

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

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

SYN-RECEIVED STATE
ESTABLISHED STATE

Enter the CLOSE-WAIT state.

FIN-WAIT-1 STATE

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

FIN-WAIT-2 STATE

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

CLOSE-WAIT STATE

Remain in the CLOSE-WAIT state.

CLOSING STATE

Remain in the CLOSING state.

LAST-ACK STATE

Remain in the LAST-ACK state.

TIME-WAIT STATE

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

and return.

USER TIMEOUT

USER TIMEOUT

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

RETRANSMISSION TIMEOUT

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

TIME-WAIT TIMEOUT

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

3.11. Glossary

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

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

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

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

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

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

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

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

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

FTP
: A file transfer protocol.

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

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

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

IMP
: The Interface Message Processor, the packet switch of the ARPANET.

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

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

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

IP
: Internet Protocol.

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

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

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

leader
: Control information at the beginning of a message or block of data. In particular, in the ARPANET, the control information on an ARPANET message at the host-IMP interface.

left sequence
: This is the next sequence number to be acknowledged by the data receiving TCP (or the lowest currently unacknowledged

sequence number) and is sometimes referred to as the left edge of the send window.

local packet
: The unit of transmission within a local network.

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

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

octet
: An eight bit byte.

Options
: An Option field may contain several options, and each option may be several octets in length. The options are used primarily in testing situations; for example, to carry timestamps. Both the Internet Protocol and TCP provide for options fields.

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

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

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

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

RCV.NXT
: receive next sequence number

RCV.UP
: receive urgent pointer

RCV.WND
: receive window

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

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

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

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

SEG.ACK
: segment acknowledgment

SEG.LEN
: segment length

SEG.PRC
: segment precedence value

SEG.SEQ
: segment sequence

SEG.UP
: segment urgent pointer field

SEG.WND
: segment window field

segment

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

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

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

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

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

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

SND.NXT
: send sequence

SND.UNA
: left sequence

SND.UP
: send urgent pointer

SND.WL1
: segment sequence number at last window update

SND.WL2
: segment acknowledgment number at last window update

SND.WND
: send window

socket

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

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

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

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

TCB.PRC
: The precedence of the connection.

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

TOS
: Type of Service, an Internet Protocol field.

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

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

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

4. Changes from RFC 793

This document obsoletes RFC 793 as well as RFC 6093 and 6528, which updated 793. In all cases, only the normative protocol specification and requirements have been incorporated into this document, and the

informational text with background and rationale has not been carried in. The informational content of those documents is still valuable in learning about and understanding TCP, and they are valid Informational references, even though their normative content has been incorporated into this document.

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

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

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

The 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.

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

The -00 revision of this document was merely a proposal and rough plan for updating RFC 793.

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

The -02 revision of this document incorporates errata that have been verified:

Errata ID 573: Reported by Bob Braden (note: This errata basically is just a reminder that RFC 1122 updates 793. Some of the associated changes are left pending to a separate revision that incorporates 1122. Bob's mention of PUSH in 793 section 2.8 was

not applicable here because that section was not part of the "functional specification". Also the 1122 text on the retransmission timeout also has been updated by subsequent RFCs, so the change here deviates from Bob's suggestion to apply the 1122 text.)
Errata ID 574: Reported by Yin Shuming
Errata ID 700: Reported by Yin Shuming
Errata ID 701: Reported by Yin Shuming
Errata ID 1283: Reported by Pei-chun Cheng
Errata ID 1561: Reported by Constantin Hagemeier
Errata ID 1562: Reported by Constantin Hagemeier
Errata ID 1564: Reported by Constantin Hagemeier
Errata ID 1565: Reported by Constantin Hagemeier
Errata ID 1571: Reported by Constantin Hagemeier
Errata ID 1572: Reported by Constantin Hagemeier
Errata ID 2296: Reported by Vishwas Manral
Errata ID 2297: Reported by Vishwas Manral
Errata ID 2298: Reported by Vishwas Manral
Errata ID 2748: Reported by Mykyta Yevstifeyev
Errata ID 2749: Reported by Mykyta Yevstifeyev
Errata ID 2934: Reported by Constantin Hagemeier
Errata ID 3213: Reported by EugnJun Yi
Errata ID 3300: Reported by Botong Huang
Errata ID 3301: Reported by Botong Huang
Note: Some verified errata were not used in this update, as they relate to sections of RFC 793 elided from this document. These include Errata ID 572, 575, and 1569.
Note: Errata ID 3602 was not applied in this revision as it is duplicative of the 1122 corrections.
There is an errata 3305 currently reported that need to be verified, held, or rejected by the ADs; it is addressing the same issue as draft-gont-tcpm-tcp-seq-validation and was not attempted to be applied to this document.

Not related to RFC 793 content, this revision also makes small tweaks to the introductory text, fixes indentation of the pseudoheader diagram, and notes that the Security Considerations should also include privacy, when this section is written.

The -03 revision of this document revises all discussion of the urgent pointer in order to comply with RFC 6093, 1122, and 1011. Since 1122 held requirements on the urgent pointer, the full list of requirements was brought into an appendix of this document, so that it can be updated as-needed.

The -04 revision of this document includes the ISN generation changes from RFC 6528.

The -05 revision of this document incorporates MSS requirements and definitions from RFC 879, 1122, and 6691, as well as option-handling requirements from RFC 1122.

TODO: Incomplete list of planned changes - these need to be added to and made more specific, as the document proceeds:

1. incorporate 1122 additions
2. point to major additional docs like 1323bis and 5681
3. incorporate relevant parts of 3168 (ECN)
4. incorporate Fernando's new number-checking fixes (if past the IESG in time)
5. point to PMTUD?
6. point to 5461 (soft errors)
7. mention 5961 state machine option
8. mention 6161 (reducing TIME-WAIT)
9. incorporate 6429 (ZWP/persist)
10. look at Tony Sabatini suggestion for describing DO field
11. clearly specify treatment of reserved bits (see TCPM thread on EDO draft April 25, 2014)
12. look at possible mention of draft-minshall-nagle (e.g. as in Linux)
13. make sure that clarifications in RFC 1011 are captured
14. per TCPM discussion, discussion of checking reserved bits may need to be altered from 793
15. MSL acronymn is defined multiple times

5. IANA Considerations

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

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

6. Security and Privacy Considerations

TODO

See RFC 6093 [7] for discussion of security considerations related to the urgent pointer field.

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

7. Acknowledgements

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

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

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

Michael Scharf
Yoshifumi Nishida
Pasi Sarolahti

On the TCPM mailing list, and at the IETF 88 meeting in Vancouver, helpful comments, critiques, and reviews were received from (listed alphebetically): David Borman, Yuchung Cheng, Martin Duke, Kevin Lahey, Kevin Mason, Matt Mathis, Hagen Paul Pfeifer, Anthony Sabatini, Joe Touch, Reji Varghese, Lloyd Wood, and Alex Zimmermann.

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

8. References

8.1. Normative References

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

8.2. Informative References

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

[3] Braden, R., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, October 1989.

[4] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, November 1990.

[5] Lahey, K., "TCP Problems with Path MTU Discovery", RFC 2923, September 2000.

[6] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, March 2007.

[7] Gont, F. and A. Yourtchenko, "On the Implementation of the TCP Urgent Mechanism", RFC 6093, January 2011.

[8] Gont, F. and S. Bellovin, "Defending against Sequence Number Attacks", RFC 6528, February 2012.

[9] Borman, D., "TCP Options and Maximum Segment Size (MSS)", RFC 6691, July 2012.

[10] Duke, M., Braden, R., Eddy, W., Blanton, E., and A. Zimmermann, "A Roadmap for Transmission Control Protocol (TCP) Specification Documents", RFC 7414, February 2015.

Appendix A. TCP Requirement Summary

This section is adapted from RFC 1122.

TODO: this needs to be seriously redone, to use 793bis section numbers instead of 1122 ones, and all 1122 requirements need to be reflected in 793bis text.

RFC EDITOR'S NOTE: 793bis in the heading below should be replaced by the number of this RFC

FEATURE	RFC1122 SECTION	TSTD	SHLD	MUST	MUSY	SHNT	MUNT	Footnote
Push flag								
Aggregate or queue un-pushed data	4.2.2.2			x				
Sender collapse successive PSH flags	4.2.2.2		x					
SEND call can specify PUSH	4.2.2.2			x				
If cannot: sender buffer indefinitely	4.2.2.2						x	
If cannot: PSH last segment	4.2.2.2	x						
Notify receiving ALP of PSH	4.2.2.2			x				1
Send max size segment when possible	4.2.2.2		x					

Window						
Treat as unsigned number	4.2.2.3	x				
Handle as 32-bit number	4.2.2.3		x			
Shrink window from right	4.2.2.16			x		
Robust against shrinking window	4.2.2.16	x				
Receiver's window closed indefinitely	4.2.2.17			x		
Sender probe zero window	4.2.2.17	x				
First probe after RTO	4.2.2.17		x			
Exponential backoff	4.2.2.17		x			
Allow window stay zero indefinitely	4.2.2.17	x				
Sender timeout OK conn with zero wind	4.2.2.17					x
Urgent Data						
Pointer indicates first non-urgent octet	4.2.2.4	x				
Arbitrary length urgent data sequence	4.2.2.4	x				
Inform ALP asynchronously of urgent data	4.2.2.4	x				1
ALP can learn if/how much urgent data Q'd	4.2.2.4	x				1
TCP Options						
Receive TCP option in any segment	4.2.2.5	x				
Ignore unsupported options	4.2.2.5	x				
Cope with illegal option length	4.2.2.5	x				
Implement sending & receiving MSS option	4.2.2.6	x				
Send MSS option unless 536	4.2.2.6		x			
Send MSS option always	4.2.2.6			x		
Send-MSS default is 536	4.2.2.6	x				
Calculate effective send seg size	4.2.2.6	x				
TCP Checksums						
Sender compute checksum	4.2.2.7	x				
Receiver check checksum	4.2.2.7	x				
ISN Selection						
Include a clock-driven ISN generator component	4.2.2.9	x				
Secure ISN generator with a PRF component	N/A		x			
Opening Connections						
Support simultaneous open attempts	4.2.2.10	x				
SYN-RCVD remembers last state	4.2.2.11	x				
Passive Open call interfere with others	4.2.2.18					x
Function: simultan. LISTENs for same port	4.2.2.18	x				
Ask IP for src address for SYN if necc.	4.2.3.7	x				
Otherwise, use local addr of conn.	4.2.3.7	x				
OPEN to broadcast/multicast IP Address	4.2.3.14					x
Silently discard seg to bcast/mcast addr	4.2.3.14	x				
Closing Connections						

RST can contain data	4.2.2.12		x			
Inform application of aborted conn	4.2.2.13	x				
Half-duplex close connections	4.2.2.13			x		
Send RST to indicate data lost	4.2.2.13		x			
In TIME-WAIT state for 2xMSL seconds	4.2.2.13	x				
Accept SYN from TIME-WAIT state	4.2.2.13			x		
Retransmissions						
Jacobson Slow Start algorithm	4.2.2.15	x				
Jacobson Congestion-Avoidance algorithm	4.2.2.15	x				
Retransmit with same IP ident	4.2.2.15			x		
Karn's algorithm	4.2.3.1	x				
Jacobson's RTO estimation alg.	4.2.3.1	x				
Exponential backoff	4.2.3.1	x				
SYN RTO calc same as data	4.2.3.1		x			
Recommended initial values and bounds	4.2.3.1		x			
Generating ACK's:						
Queue out-of-order segments	4.2.2.20		x			
Process all Q'd before send ACK	4.2.2.20	x				
Send ACK for out-of-order segment	4.2.2.21			x		
Delayed ACK's	4.2.3.2		x			
Delay < 0.5 seconds	4.2.3.2	x				
Every 2nd full-sized segment ACK'd	4.2.3.2	x				
Receiver SWS-Avoidance Algorithm	4.2.3.3	x				
Sending data						
Configurable TTL	4.2.2.19	x				
Sender SWS-Avoidance Algorithm	4.2.3.4	x				
Nagle algorithm	4.2.3.4		x			
Application can disable Nagle algorithm	4.2.3.4	x				
Connection Failures:						
Negative advice to IP on R1 retxs	4.2.3.5	x				
Close connection on R2 retxs	4.2.3.5	x				
ALP can set R2	4.2.3.5	x				1
Inform ALP of R1<=retxs<R2	4.2.3.5		x			1
Recommended values for R1, R2	4.2.3.5		x			
Same mechanism for SYNs	4.2.3.5	x				
R2 at least 3 minutes for SYN	4.2.3.5	x				
Send Keep-alive Packets:	4.2.3.6			x		
- Application can request	4.2.3.6	x				
- Default is "off"	4.2.3.6	x				
- Only send if idle for interval	4.2.3.6	x				
- Interval configurable	4.2.3.6	x				
- Default at least 2 hrs.	4.2.3.6	x				
- Tolerant of lost ACK's	4.2.3.6	x				

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

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

Author's Address

Wesley M. Eddy (editor)
MTI Systems
US

Email: wes@mti-systems.com

TCPM Working Group
Internet-Draft
Obsoletes: 2861 (if approved)
Intended status: Experimental
Expires: December 27, 2015

G. Fairhurst
A. Sathiaseelan
R. Secchi
University of Aberdeen

June 25, 2015

Updating TCP to support Rate-Limited Traffic
draft-ietf-tcpm-newcwv-13

Abstract

This document provides a mechanism to address issues that arise when TCP is used for traffic that exhibits periods where the sending rate is limited by the application rather than the congestion window. It provides an experimental update to TCP that allows a TCP sender to restart quickly following a rate-limited interval. This method is expected to benefit applications that send rate-limited traffic using TCP, while also providing an appropriate response if congestion is experienced.

It also evaluates the Experimental specification of TCP Congestion Window Validation, CWV, defined in RFC 2861, and concludes that RFC 2861 sought to address important issues, but failed to deliver a widely used solution. This document therefore recommends that the status of RFC 2861 is moved from Experimental to Historic, and that it is replaced by the current specification.

Status of This Memo

Table of Contents

1. Introduction 3
1.1. Implementation of new CWV 5
1.2. Standards Status of this Document 5
2. Reviewing experience with TCP-CWV 5
3. Terminology 7
4.1. Initialisation 8
4.2. Estimating the validated capacity supported by a path . . 8
4.3. Preserving cwnd during a rate-limited period. 10
4.4. TCP congestion control during the non-validated phase . . 11
4.4.1. Response to congestion in the non-validated phase . . 12
4.4.2. Sender burst control during the non-validated phase . 13
4.4.3. Adjustment at the end of the Non-Validated Period (NVP) 14
4.5. Examples of Implementation 15
4.5.1. Implementing the pipeACK measurement 15
4.5.2. Measurement of the NVP and pipeACK samples 16
4.5.3. Implementing detection of the cwnd-limited condition 16
5. Determining a safe period to preserve cwnd 17
6. Security Considerations 18
7. IANA Considerations 18
8. Acknowledgments 18
9. Author Notes 18
9.1. Other related work 18
10. Revision notes 20
11. References 24
11.1. Normative References 24
11.2. Informative References 25
Authors' Addresses 26

1. Introduction

TCP is used for traffic with a range of application behaviours. The TCP congestion window (cwnd) controls the maximum number of unacknowledged packets/bytes that a TCP flow may have in the network at any time, a value known as the FlightSize [RFC5681]. FlightSize is a measure of the volume of data that is unacknowledged at a specific time. A bulk application will always have data available to transmit. The rate at which it sends is therefore limited by the maximum permitted by the receiver advertised window and the sender congestion window (cwnd). The FlightSize of a bulk flow increases with the cwnd, and tracks the volume of data acknowledged in the last Round Trip Time (RTT).

In contrast, a rate-limited application will experience periods when the sender is either idle or is unable to send at the maximum rate permitted by the cwnd. In this case, the volume of data sent (FlightSize) can change significantly from one RTT to another, and can be much less than the cwnd. Hence, it is possible that the FlightSize could significantly exceed the recently used capacity. The update in this document targets the operation of TCP in such rate-limited cases.

Standard TCP [RFC5681] states that a TCP sender SHOULD set cwnd to no more than the Restart Window (RW) before beginning transmission, if the TCP sender has not sent data in an interval exceeding the retransmission timeout, i.e., when an application becomes idle. [RFC2861] noted that this TCP behaviour was not always observed in current implementations. Experiments [Bis08] confirm this to still be the case.

Congestion Window Validation, CWV, introduced the terminology of "application limited periods". RFC2861 describes any time that an application limits the sending rate, rather than being limited by the transport, as "rate-limited". This update improves support for applications that vary their transmission rate, either with (short) idle periods between transmission or by changing the rate at which the application sends. These applications are characterised by the TCP FlightSize often being less than cwnd. Many Internet applications exhibit this behaviour, including web browsing, http-based adaptive streaming, applications that support query/response type protocols, network file sharing, and live video transmission. Many such applications currently avoid using long-lived (persistent) TCP connections (e.g., [RFC7230] servers typically support persistent HTTP connections, but do not enable this by default). Such applications often instead either use a succession of short TCP transfers or use UDP.

Standard TCP does not impose additional restrictions on the growth of the congestion window when a TCP sender is unable to send at the maximum rate allowed by the cwnd. In this case, the rate-limited sender may grow a cwnd far beyond that corresponding to the current transmit rate, resulting in a value that does not reflect current information about the state of the network path the flow is using. Use of such an invalid cwnd may result in reduced application performance and/or could significantly contribute to network congestion.

[RFC2861] proposed a solution to these issues in an experimental method known as CWV. CWV was intended to help reduce cases where TCP accumulated an invalid (inappropriately large) cwnd. The use and drawbacks of using the CWV algorithm in RFC 2861 with an application are discussed in Section 2.

Section 3 defines relevant terminology.

Section 4 specifies an alternative to CWV that seeks to address the same issues, but does so in a way that is expected to mitigate the impact on an application that varies its sending rate. The updated method applies to the rate-limited conditions (including both application-limited and idle senders).

The goals of this update are:

- To not change the behaviour of a TCP sender that performs bulk transfers that fully use the cwnd.
- To provide a method that co-exists with Standard TCP and other flows that use this updated method.
- To reduce transfer latency for applications that change their rate over short intervals of time.
- To avoid a TCP sender growing a large "non-validated" cwnd, when it has not recently sent using this cwnd.
- To remove the incentive for ad-hoc application or network stack methods (such as "padding") solely to maintain a large cwnd for future transmission.
- To provide an incentive for the use of long-lived connections, rather than a succession of short-lived flows, benefiting both the flows and other flows sharing the network path when actual congestion is encountered.

Section 5 describes the rationale for selecting the safe period to preserve the cwnd.

1.1. Implementation of new CWV

The method specified in Section 4 of this document is a sender-side only change to the the TCP congestion control behaviour of TCP.

The method creates a new protocol state, and requires a sender to determine when the cwnd is validated or non-validated to control the entry and exit from this state Section 4.3. It defines how a TCP sender manages the growth of the cwnd using the set of rules defined in Section 4.

Implementation of this specification requires an implementor to define a method to measure the available capacity using the pipeACK samples. The details of this measurement are implementation-specific. An example is provided in Section 4.5.1, but other methods are permitted. A sender also needs to provide a method to determine when it becomes cwnd-limited. Implementation of this may require consideration of other TCP methods (see Section 4.5.3).

A sender is also recommended to provide a method that controls the maximum burst size, Section 4.4.2. However, implementors are allowed flexibility in how this method is implemented and the choice of an appropriate method is expected to depend on the way in which the sender stack implements other TCP methods (such as TCP Segment Offload, TSO).

1.2. Standards Status of this Document

The document obsoletes the methods described in [RFC2861]. It recommends a set of mechanisms, including the use of pacing during a non-validated period. The updated mechanisms are intended to have a less aggressive congestion impact than would be exhibited by a standard TCP sender.

The specification in this draft is classified as "Experimental" pending experience with deployed implementations of the methods.

2. Reviewing experience with TCP-CWV

[RFC2861] described a simple modification to the TCP congestion control algorithm that decayed the cwnd after the transition to a "sufficiently-long" idle period. This used the slow-start threshold (ssthresh) to save information about the previous value of the congestion window. The approach relaxed the standard TCP behaviour [RFC5681] for an idle session, intended to improve application

performance. CWV also modified the behaviour when a sender transmitted at a rate less than allowed by cwnd.

[RFC2861] proposed two set of responses, one after an "application-limited" and one after an "idle period". Although this distinction was argued, in practice differentiating the two conditions was found problematic in actual networks (e.g., [Bis10]). While this offers predictable performance for long on-off periods (>>1 RTT), or slowly varying rate-based traffic, the performance could be unpredictable for variable-rate traffic and depended both upon whether an accurate RTT had been obtained and the pattern of application traffic relative to the measured RTT.

Many applications can and often do vary their transmission over a wide range of rates. Using [RFC2861] such applications often experienced varying performance, which made it hard for application developers to predict the TCP latency even when using a path with stable network characteristics. We argue that an attempt to classify application behaviour as application-limited or idle is problematic and also inappropriate. This document therefore explicitly avoids trying to differentiate these two cases, instead treating all rate-limited traffic uniformly.

[RFC2861] has been implemented in some mainstream operating systems as the default behaviour [Bis08]. Analysis (e.g., [Bis10] [Fai12]) has shown that a TCP sender using CWV is able to use available capacity on a shared path after an idle period. This can benefit variable-rate applications, especially over long delay paths, when compared to the slow-start restart specified by standard TCP. However, CWV would only benefit an application if the idle period were less than several Retransmission Time Out (RTO) intervals [RFC6298], since the behaviour would otherwise be the same as for standard TCP, which resets the cwnd to the TCP Restart Window after this period.

To enable better performance for variable-rate applications with TCP, some operating systems have chosen to support non-standard methods, or applications have resorted to "padding" streams by sending dummy data to maintain their sending rate when they have no data to transmit. Although transmitting redundant data across a network path provides good evidence that the path can sustain data at the offered rate, padding also consumes network capacity and reduces the opportunity for congestion-free statistical multiplexing. For variable-rate flows, the benefits of statistical multiplexing can be significant and it is therefore a goal to find a viable alternative to padding streams.

Experience with [RFC2861] suggests that although the CWV method benefited the network in a rate-limited scenario (reducing the probability of network congestion), the behaviour was too conservative for many common rate-limited applications. This mechanism did not therefore offer the desirable increase in application performance for rate-limited applications and it is unclear whether applications actually use this mechanism in the general Internet.

It is therefore concluded that CWV, as defined in [RFC2861], was often a poor solution for many rate-limited applications. It had the correct motivation, but had the wrong approach to solving this problem.

3. Terminology

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

The document assumes familiarity with the terminology of TCP congestion control [RFC5681].

The following additional terminology is introduced in this document:

cwnd-limited: A TCP flow that has sent the maximum number of segments permitted by the cwnd, where the application utilises the allowed sending rate (see Section 4.5.3).

pipeACK sample: A measure of the volume of data acknowledged by the network within an RTT.

pipeACK variable: A variable that measures the available capacity using the set of pipeACK samples.

pipeACK Sampling Period: The maximum period that a measured pipeACK sample may influence the pipeACK variable.

Non-validated phase: The phase where the cwnd reflects a previous measurement of the available path capacity.

Non-validated period, NVP: The maximum period for which cwnd is preserved in the non-validated phase.

Rate-limited: A TCP flow that does not consume more than one half of cwnd, and hence operates in the non-validated phase. This includes periods when an application is either idle or chooses to send at a rate less than the maximum permitted by the cwnd.

Validated phase: The phase where the cwnd reflects a current estimate of the available path capacity.

4. A New Congestion Window Validation method

This section proposes an update to the TCP congestion control behaviour during a rate-limited interval. This new method intentionally does not differentiate between times when the sender has become idle or chooses to send at a rate less than the maximum allowed by the cwnd.

The period where actual usage is less than allowed by cwnd, is named the non-validated phase. The update allows an application in the non-validated phase to resume transmission at a previous rate without incurring the delay of slow-start. However, if the TCP sender experiences congestion using the preserved cwnd, it is required to immediately reset the cwnd to an appropriate value specified by the method. If a sender does not take advantage of the preserved cwnd within the Non-validated period, NVP, the value of cwnd is reduced, ensuring the value better reflects the capacity that was recently actually used.

It is expected that this update will satisfy the requirements of many rate-limited applications and at the same time provide an appropriate method for use in the Internet. New-CWV reduces this incentive for an application to send "padding" data simply to keep transport congestion state.

The method is specified in following subsections and is expected to encourage applications and TCP stacks to use standards-based congestion control methods. It may also encourage the use of long-lived connections where this offers benefit (such as persistent http).

4.1. Initialisation

A sender starts a TCP connection in the validated phase and initialises the pipeACK variable to the "undefined" value. This value inhibits use of the value in cwnd calculations.

4.2. Estimating the validated capacity supported by a path

[RFC6675] defines a variable, FlightSize, that indicates the instantaneous amount of data that has been sent, but not cumulatively acknowledged. In this method a new variable "pipeACK" is introduced to measure the acknowledged size of the network pipe. This is used to determine if the sender has validated the cwnd. pipeACK differs

from FlightSize in that it is evaluated over a window of acknowledged data, rather than reflecting the amount of data outstanding.

A sender determines a pipeACK sample by measuring the volume of data that was acknowledged by the network over the period of a measured Round Trip Time (RTT). Using the variables defined in [RFC6675], a value could be measured by caching the value of HighACK and after one RTT measuring the difference between the cached HighACK value and the current HighACK value. A sender MAY count TCP DupACKs that acknowledge new data when collecting the pipeACK sample. Other equivalent methods may be used.

A sender is not required to continuously update the pipeACK variable after each received ACK, but SHOULD perform a pipeACK sample at least once per RTT when it has sent unacknowledged segments.

The pipeACK variable MAY consider multiple pipeACK samples over the pipeACK Sampling Period. The value of the pipeACK variable MUST NOT exceed the maximum (highest value) within the sampling period. This specification defines the pipeACK Sampling Period as Max(3*RTT, 1 second). This period enables a sender to compensate for large fluctuations in the sending rate, where there may be pauses in transmission, and allows the pipeACK variable to reflect the largest recently measured pipeACK sample.

When no measurements are available (e.g., a sender that has just started transmission or immediately after loss recovery), the pipeACK variable is set to the "undefined value". This value is used to inhibit entering the non-validated phase until the first new measurement of a pipeACK sample. (Section 4.5 provides examples of implementation.)

The pipeACK variable MUST NOT be updated during TCP Fast Recovery. That is, the sender stops collecting pipeACK samples during loss recovery. The method RECOMMENDS enabling the TCP SACK option [RFC2018] and RECOMMENDS the method defined in [RFC6675] to recover missing segments. This allows the sender to more accurately determine the number of missing bytes during the loss recovery phase, and using this method will result in a more appropriate cwnd following loss.

NOTE: The use of pipeACK rather than FlightSize can change the behaviour of a TCP when a sender does not always have data available to send. One example arises when there is a pause in transmission after sending a sequence of many packets, and the sender experiences loss at or near the end of its transmission sequence. In this case, the TCP flow may have used a significant amount of capacity just prior to the loss (which would be reflected in the volume of data

acknowledged, recorded in the pipeACK variable), but at the actual time of loss the number of unacknowledged packets in flight (at the end of the sequence) may be small, i.e., there is a small FlightSize. After loss recovery, the sender resets its congestion control state.

[Fai12] explored the benefits of different responses to congestion for application-limited streams. If the response is based only on the Loss FlightSize, the sender would assign a small cwnd and ssthresh, based only on the volume of data sent after the loss. When the sender next starts to transmit it can incur may RTTs of delay in slow start before it reacquires its previous rate. When the pipeACK value is also usedto calculate the cwnd and ssthresh (as specified in this update in Section 4.4.1), the sender can use a value that also reflects the recently used capacity before the loss. This prevents a variable-rate application from being unduly penalised. When the sender resumes, it starts at one half its previous rate, similar to the behaviour of a bulk TCP flow [Hos15]. To ensure an appropriate reaction to on-going congestion, this method requires that the pipeACK variable is reset after it is used in this way.

4.3. Preserving cwnd during a rate-limited period.

The updated method creates a new TCP sender phase that captures whether the cwnd reflects a validated or non-validated value. The phases are defined as:

- Validated phase: pipeACK >=(1/2)*cwnd, or pipeACK is undefined (i.e., at the start or directly after loss recovery). This is the normal phase, where cwnd is expected to be an approximate indication of the capacity currently available along the network path, and the standard methods are used to increase cwnd (currently [RFC5681]).

- Non-validated phase: pipeACK <(1/2)*cwnd. This is the phase where the cwnd has a value based on a previous measurement of the available capacity, and the usage of this capacity has not been validated in the pipeACK Sampling Period. That is, when it is not known whether the cwnd reflects the currently available capacity along the network path. The mechanisms to be used in this phase seek to determine a safe value for cwnd and an appropriate reaction to congestion.

Note: A threshold is needed to determine whether a sender is in the validated or non-validated phase. A standard TCP sender in slow-start is permitted to double its FlightSize from one RTT to the next. This motivated the choice of a threshold value of 1/2. This threshold ensures a sender does not further increase the cwnd as long as the FlightSize is less than (1/2*cwnd). Furthermore, a sender

with a FlightSize less than (1/2*cwnd) may in the next RTT be permitted by the cwnd to send at a rate that more than doubles the FlightSize, and hence this case needs to be regarded as non-validated and a sender therefore needs to employ additional mechanisms while in this phase.

4.4. TCP congestion control during the non-validated phase

A TCP sender implementing this specification MUST enter the non-validated phase when the pipeACK is less than (1/2)*cwnd. (The note at the end of section 4.4.1 describes why pipeACK<=(1/2)*cwnd is expected to be a safe value.)

A TCP sender that enters the non-validated phase preserves the cwnd (i.e., the cwnd only increases after a sender fully uses the cwnd in this phase, otherwise the cwnd neither grows nor reduces). The phase is concluded when the sender transmits sufficient data so that pipeACK > (1/2)*cwnd (i.e., the sender is no longer rate-limited), or when the sender receives an indication of congestion.

After a fixed period of time (the non-validated period, NVP), the sender adjusts the cwnd Section 4.4.3). The NVP SHOULD NOT exceed 5 minutes.Section 5 discusses the rationale for choosing a safe value for this period.

The behaviour in the non-validated phase is specified as:

- A sender determines whether to increase the cwnd based upon whether it is cwnd-limited (see Section 4.5.3):
  - A sender that is cwnd-limited MAY use the standard TCP method to increase cwnd (i.e., a TCP sender that fully utilises the cwnd is permitted to increase cwnd each received ACK using standard methods).
  - A sender that is not cwnd-limited MUST NOT increase the cwnd when ACK packets are received in this phase (i.e., needs to avoid growing the cwnd when it has not recently sent using the current size of cwnd).
- If the sender receives an indication of congestion while in the non-validated phase (i.e., detects loss), the sender MUST exit the non-validated phase (reducing the cwnd as defined in Section 4.4.1).
- If the Retransmission Time Out (RTO) expires while in the non-validated phase, the sender MUST exit the non-validated phase. It then resumes using the standard TCP RTO mechanism [RFC5681].

- o A sender with a pipeACK variable greater than (1/2)*cwnd SHOULD enter the validated phase. (A rate-limited sender will not normally be impacted by whether it is in a validated or non-validated phase, since it will normally not increase FlightSize to use the entire cwnd. However, a change to the validated phase will release the sender from constraints on the growth of cwnd, and result in using the standard congestion response.)

The cwnd-limited behaviour may be triggered during a transient condition that occurs when a sender is in the non-validated phase and receives an ACK that acknowledges received data, the cwnd was fully utilised, and more data is awaiting transmission than may be sent with the current cwnd. The sender MAY then use the standard method to increase the cwnd. (Note, if the sender succeeds in sending these new segments, the updated cwnd and pipeACK variables will eventually result in a transition to the validated phase.)

4.4.1. Response to congestion in the non-validated phase

Reception of congestion feedback while in the non-validated phase is interpreted as an indication that it was inappropriate for the sender to use the preserved cwnd. The sender is therefore required to quickly reduce the rate to avoid further congestion. Since the cwnd does not have a validated value, a new cwnd value needs to be selected based on the utilised rate.

A sender that detects a packet-drop MUST record the current FlightSize in the variable LossFlightSize and MUST calculate a safe cwnd for loss recovery using the method below:

```
cwnd = (Max(pipeACK,LossFlightSize))/2.
```

The pipeACK value is not updated during loss recovery (see Section 4.2). If there is a valid pipeACK value, the new cwnd is adjusted to reflect that a non-validated cwnd may be larger than the actual FlightSize, or recently used FlightSize (recorded in pipeACK). The updated cwnd therefore prevents overshoot by a sender significantly increasing its transmission rate during the recovery period.

At the end of the recovery phase, the TCP sender MUST reset the cwnd using the method below:

```
cwnd = (Max(pipeACK,LossFlightSize) - R)/2.
```

Where R is the volume of data that was successfully retransmitted during the recovery phase. This corresponds to segments

retransmitted and considered lost by the pipe estimation algorithm at the end of recovery. It does not include the additional cost of multiple retransmission of the same data. The loss of segments indicates that the path capacity was exceeded by at least R, and hence the calculated cwnd is reduced by at least R before the window is halved.

The calculated cwnd value MUST NOT be reduced below 1 TCP Maximum Segment Size (MSS).

After completing the loss recovery phase, the sender MUST re-initialise the pipeACK variable to the "undefined" value. This ensures that standard TCP methods are used immediately after completing loss recovery until a new pipeACK value can be determined.

The ssthresh is adjusted using the standard TCP method (Step 6 in Section 3.2 of RFC 5681 assigns the ssthresh a value equal to cwnd at the end of the loss recovery).

Note: The adjustment by reducing cwnd by the volume of data not sent (R) follows the method proposed for Jump Start [Liu07]. The inclusion of the term R makes the adjustment more conservative than standard TCP. This is required, since a sender in the non-validated state may increase the rate more than a standard TCP would have done relative to what was sent in the last RTT (i.e., more than doubled the number of segments in flight relative to what it sent in the last RTT). The additional reduction after congestion is beneficial when the LossFlightSize has significantly overshot the available path capacity incurring significant loss (e.g., following a change of path characteristics or when additional traffic has taken a larger share of the network bottleneck during a period when the sender transmits less).

Note: The pipeACK value is only valid during a non-validated phase, and therefore this does not exceed cwnd/2. If LossFlightSize and R were small, then this can result in the final cwnd after loss recovery being at most one quarter of the cwnd on detection of congestion. This reduction is conservative, and pipeACK is then reset to undefined, hence cwnd updates after a congestion event do not depend upon the pipeACK history before congestion was detected.

4.4.2. Sender burst control during the non-validated phase

TCP congestion control allows a sender to accumulate a cwnd that would allow it to send a burst of segments with a total size up to the difference between the FlightsSize and cwnd. Such bursts can impact other flows that share a network bottleneck and/or may induce congestion when buffering is limited.

Various methods have been proposed to control the sender burstiness [Hug01], [All05]. For example, TCP can limit the number of new segments it sends per received ACK. This is effective when a flow of ACKs is received, but can not be used to control a sender that has not send appreciable data in the previous RTT [All05].

This document recommends using a method to avoid line-rate bursts after an idle or rate-limited interval when there is less reliable information about the capacity of the network path: A TCP sender in the non-validated phase SHOULD control the maximum burst size, e.g., using a rate-based pacing algorithm in which a sender paces out the cwnd over its estimate of the RTT, or some other method, to prevent many segments being transmitted contiguously at line-rate. The most appropriate method(s) to implement pacing depend on the design of the TCP/IP stack, speed of interface and whether hardware support (such as TCP Segment Offload, TSO) is used. The present document does not recommend any specific method.

4.4.3. Adjustment at the end of the Non-Validated Period (NVP)

An application that remains in the non-validated phase for a period greater than the NVP is required to adjust its congestion control state. If the sender exits the non-validated phase after this period, it MUST update the ssthresh:

ssthresh = max(ssthresh, 3*cwnd/4).

(This adjustment of ssthresh ensures that the sender records that it has safely sustained the present rate. The change is beneficial to rate-limited flows that encounter occasional congestion, and could otherwise suffer an unwanted additional delay in recovering the sending rate.)

The sender MUST then update cwnd to be not greater than:

cwnd = max((1/2)*cwnd, IW).

Where IW is the appropriate TCP initial window, used by the TCP sender (e.g., [RFC5681]).

Note: These cwnd and ssthresh adjustments cause the sender to enter slow-start (since ssthresh > cwnd). This adjustment ensures that the sender responds conservatively after remaining in the non-validated phase for more than the non-validated period. In this case, it reduces the cwnd by a factor of two from the preserved value. This adjustment is helpful when flows accumulate but do not use a large cwnd, and seeks to mitigate the impact when these flows later resume

transmission. This could for instance mitigate the impact if multiple high-rate application flows were to become idle over an extended period of time and then were simultaneously awakened by an external event.

4.5. Examples of Implementation

This section provides informative examples of implementation methods. Implementations may choose to use other methods that comply with the normative requirements.

4.5.1. Implementing the pipeACK measurement

A pipeACK sample may be measured once each RTT. This reduces the sender processing burden for calculating after each acknowledgement and also reduces storage requirements at the sender.

Since application behaviour can be bursty using CWV, it may be desirable to implement a maximum filter to accumulate the measured values so that the pipeACK variable records the largest pipeACK sample within the pipeACK Sampling Period. One simple way to implement this is to divide the pipeACK Sampling Period into several (e.g., 5) equal length measurement periods. The sender then records the start time for each measurement period and the highest measured pipeACK sample. At the end of the measurement period, any measurement(s) that are older than the pipeACK Sampling Period are discarded. The pipeACK variable is then assigned the largest of the set of the highest measured values.

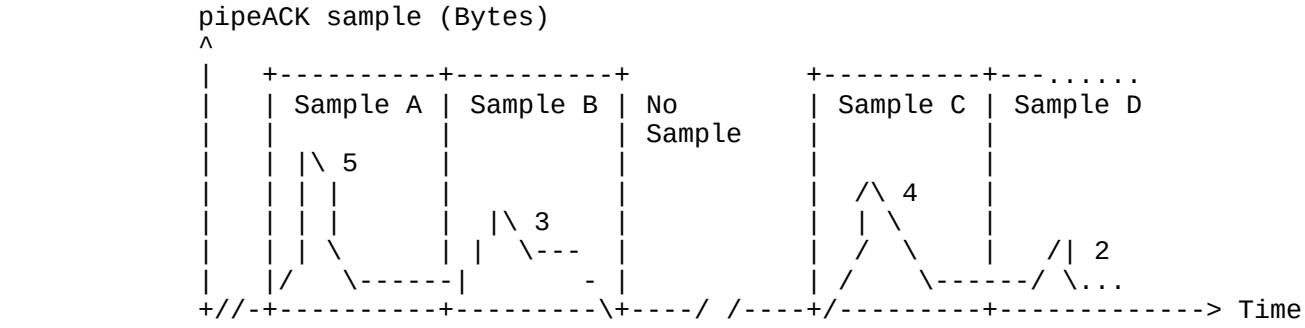


Figure 1: Example of measuring pipeACK samples

Figure 1 shows an example of how measurement samples may be collected. At the time represented by the figure new samples are being accumulated into sample D. Three previous samples also fall within the pipeACK Sampling Period: A, B, and C. There was also a period of inactivity between samples B and C during which no measurements were taken (because no new data segments were acknowledged). The current value of the pipeACK variable will be 5, the maximum across all samples. During this period, the pipeACK samples may be regarded as zero, and hence do not contribute to the calculated pipeACK value.

After one further measurement period, Sample A will be discarded, since it then is older than the pipeACK Sampling Period and the pipeACK variable will be recalculated, Its value will be the larger of Sample C or the final value accumulated in Sample D.

4.5.2. Measurement of the NVP and pipeACK samples

The mechanism requires a number of measurements of time. These measurements could be implemented using protocol timers, but do not necessarily require a new timer to be implemented. Avoiding the use of dedicated timers can save operating system resources, especially when there may be large numbers of TCP flows.

The NVP could be measured by recording a timestamp when the sender enters the non-validated phase. Each time a sender transmits a new segment, this timestamp can be used to determine if the NVP has expired. If the measured period exceeds the NVP, the sender can then take into account how many units of the NVP have passed and make one reduction (defined in Section 4.4.3) for each NVP.

Similarly, the time measurements for collecting pipeACK samples and determining the Sampling Period could be derived by using a timestamp to record when each sample was measured, and to use this to calculate how much time has passed when each new ACK is received.

4.5.3. Implementing detection of the cwnd-limited condition

A sender needs to implement a method that detects the cwnd-limited condition (see Section 4.4). This detects a condition where a sender in the non-validated phase receives an ACK, but the size of cwnd prevents sending more new data.

In simple terms, this condition is true only when the FlightSize of a TCP sender is equal to or larger than the current cwnd. However, an implementation also needs to consider constraints on the way in which the cwnd variable can be used, for instance implementations need to support other TCP methods such as the Nagle Algorithm and TCP Segment

Offload (TSO) that also use cwnd to control transmission. These other methods can result in a sender becoming cwnd-limited when the cwnd is nearly, rather than completely, equal to the FlightSize.

5. Determining a safe period to preserve cwnd

This section documents the rationale for selecting the maximum period that cwnd may be preserved, known as the NVP.

Limiting the period that cwnd may be preserved avoids undesirable side effects that would result if the cwnd were to be kept unnecessarily high for an arbitrary long period, which was a part of the problem that CWV originally attempted to address. The period a sender may safely preserve the cwnd, is a function of the period that a network path is expected to sustain the capacity reflected by cwnd. There is no ideal choice for this time.

A period of five minutes was chosen for this NVP. This is a compromise that was larger than the idle intervals of common applications, but not sufficiently larger than the period for which the capacity of an Internet path may commonly be regarded as stable. The capacity of wired networks is usually relatively stable for periods of several minutes and that load stability increases with the capacity. This suggests that cwnd may be preserved for at least a few minutes.

There are cases where the TCP throughput exhibits significant variability over a time less than five minutes. Examples could include wireless topologies, where TCP rate variations may fluctuate on the order of a few seconds as a consequence of medium access protocol instabilities. Mobility changes may also impact TCP performance over short time scales. Senders that observe such rapid changes in the path characteristic may also experience increased congestion with the new method, however such variation would likely also impact TCP’s behaviour when supporting interactive and bulk applications.

Routing algorithms may change the the network path that is used by a transport. Although a change of path can in turn disrupt the RTT measurement and may result in a change of the capacity available to a TCP connection, we assume these path changes do not usually occur frequently (compared to a time frame of a few minutes).

The value of five minutes is therefore expected to be sufficient for most current applications. Simulation studies (e.g., [Bis11]) also suggest that for many practical applications, the performance using this value will not be significantly different to that observed using a non-standard method that does not reset the cwnd after idle.

Finally, other TCP sender mechanisms have used a 5 minute timer, and there could be simplifications in some implementations by reusing the same interval. TCP defines a default user timeout of 5 minutes [RFC0793] i.e., how long transmitted data may remain unacknowledged before a connection is forcefully closed.

6. Security Considerations

General security considerations concerning TCP congestion control are discussed in [RFC5681]. This document describes an algorithm that updates one aspect of the congestion control procedures, and so the considerations described in RFC 5681 also apply to this algorithm.

7. IANA Considerations

There are no IANA considerations.

8. Acknowledgments

This document was produced by the TCP Maintenance and Minor Extensions (tcpm) working group.

The authors acknowledge the contributions of Dr I Biswas, Dr Ziaul Hossain in supporting the evaluation of CWV and for their help in developing the mechanisms proposed in this draft. We also acknowledge comments received from the Internet Congestion Control Research Group, in particular Yuchung Cheng, Mirja Kuehlewind, Joe Touch, and Mark Allman. This work was part-funded by the European Community under its Seventh Framework Programme through the Reducing Internet Transport Latency (RITE) project (ICT-317700).

9. Author Notes

RFC-Editor note: please remove this section prior to publication.

9.1. Other related work

RFC-Editor note: please remove this section prior to publication.

There are several issues to be discussed more widely:

- There are potential interactions with the Experimental update in RFC 6928 that raises the TCP initial Window to ten segments, do these cases need to be elaborated?

This relates to the Experimental specification for increasing the TCP IW defined in RFC 6928.

The two methods have different functions and different response to loss/congestion.

RFC 6928 proposes an experimental update to TCP that would increase the IW to ten segments. This would allow faster opening of the cwnd, and also a large (same size) restart window. This approach is based on the assumption that many forward paths can sustain bursts of up to ten segments without (appreciable) loss. Such a significant increase in cwnd must be matched with an equally large reduction of cwnd if loss/congestion is detected, and such a congestion indication is likely to require future use of IW=10 to be disabled for this path for some time. This guards against the unwanted behaviour of a series of short flows continuously flooding a network path without network congestion feedback.

In contrast, this document proposes an update with a rationale that relies on recent previous path history to select an appropriate cwnd after restart.

The behaviour differs in three ways:

1) For applications that send little initially, new-cwv may constrain more than RFC 6928, but would not require the connection to reset any path information when a restart incurred loss. In contrast, new-cwv would allow the TCP connection to preserve the cached cwnd, any loss, would impact cwnd, but not impact other flows.

2) For applications that utilise more capacity than provided by a cwnd of 10 segments, this method would permit a larger restart window compared to a restart using the method in RFC 6928. This is justified by the recent path history.

3) new-CWV is attended to also be used for rate-limited applications, where the application sends, but does not seek to fully utilise the cwnd. In this case, new-cwv constrains the cwnd to that justified by the recent path history. The performance trade-offs are hence different, and it would be possible to enable new-cwv when also using the method in RFC 6928, and yield benefits.

- There is potential overlap with the Laminar proposal (draft-mathis-tcpm-tcp-laminar)

The current draft was intended as a standards-track update to TCP, rather than a new transport variant. At least, it would be good to understand how the two interact and whether there is a possibility of a single method.

o There is potential performance loss in loss of a short burst (off list with M Allman)

A sender can transmit several segments then become idle. If the first set of segments are all Acknowledged, the ssthresh collapses to a small value (no new data is sent by the idle sender). Loss of the later data results in congestion (e.g., maybe a RED drop or some other cause, rather than the maximum rate of this flow). When the sender performs loss recovery it may have an appreciable pipeACK and cwnd, but a very low FlightSize - the Standard algorithm therefore results in an unusually low cwnd ((1/2)* FlightSize).

A constant rate flow would have maintained a FlightSize appropriate to pipeACK (cwnd, if it is a bulk flow).

This could be fixed by adding a new state variable? It could also be argued this is a corner case (e.g., loss of only the last segments would have resulted in RTO), the impact could be significant.

o There is potential interaction with TCP Control Block Sharing(M Welzl)

An application that is non-validated can accumulate a cwnd that is larger than the actual capacity. Is this a fair value to use in TCB sharing?

We propose that TCB sharing should use the pipeACK in place of cwnd when a TCP sender is in the Non-validated phase. This value better reflects the capacity that the flow has utilised in the network path.

10. Revision notes

RFC-Editor note: please remove this section prior to publication.

Draft 03 was submitted to ICCRG to receive comments and feedback.

Draft 04 contained the first set of clarifications after feedback:

- Changed name to application limited and used the term rate-limited in all places.
- Added justification and many minor changes suggested on the list.
- Added text to tie-in with more accurate ECN marking.
- Added ref to Hug01

Draft 05 contained various updates:

- New text to redefine how to measure the acknowledged pipe, differentiating this from the FlightSize, and hence avoiding previous issues with infrequent large bursts of data not being validated. A key point new feature is that pipeACK only triggers leaving the NVP after the size of the pipe has been acknowledged. This removed the need for hysteresis.
- Reduction values were changed to 1/2, following analysis of suggestions from ICCRG. This also sets the "target" cwnd as twice the used rate for non-validated case.
- Introduced a symbolic name (NVP) to denote the 5 minute period.

Draft 06 contained various updates:

- Required reset of pipeACK after congestion.
- Added comment on the effect of congestion after a short burst (M. Allman).
- Correction of minor Typos.

WG draft 00 contained various updates:

- Updated initialisation of pipeACK to maximum value.
- Added note on intended status still to be determined.

WG draft 01 contained:

- Added corrections from Richard Scheffenegger.
- Raffaello Secchi added to the mechanism, based on implementation experience.

- Removed that the requirement for the method to use TCP SACK option
- Although it may be desirable to use SACK, this is not essential to the algorithm.
- Added the notion of the sampling period to accommodate large rate variations and ensure that the method is stable. This algorithm to be validated through implementation.

WG draft 02 contained:

- Clarified language around pipeACK variable and pipeACK sample - Feedback from Aris Angelogiannopoulos.

WG draft 03 contained:

- Editorial corrections - Feedback from Anna Brunstrom.
- An adjustment to the procedure at the start and end of Reoloss recovery to align the two equations.
- Further clarification of the "undefined" value of the pipeACK variable.

WG draft 04 contained:

- Editorial corrections.
- Introduced the "cwnd-limited" term.
- An adjustment to the procedure at the start of a cwnd-limited phase - the new text is intended to ensure that new-cwv is not unnecessarily more conservative than standard TCP when the flow is cwnd-limited. This resolves two issues: first it prevents pathologies in which pipeACK increases slowly and erratically. It also ensures that performance of bulk applications is not significantly impacted when using the method.
- Clearly identifies that pacing (or equivalent) is requiring during the NVP to control burstiness. New section added.

WG draft 05 contained:

- Clarification to first two bullets in Section 4.4 describing cwnd-limited, to explain these are really alternates to the same case.
- Section giving implementation examples was restructured to clarify there are two methods described.

- Cross References to sections updated - thanks to comments from Martin Winbjoerk and Tim Wicinski.

WG draft 06 contained:

- The section giving implementation examples was restructured to clarify there are two methods described.
- Justification of design decisions.
- Re-organised text to improve clarity of argument.

WG draft 07 contained:

- Updated publication date.
- Text on noting that cwnd shouldn’t ever be made negative.
- Updated text on ECN to clarify the process where R is a reduction based on ECN marks.

WG draft 08 contained:

- Removed description of how to use Accurate ECN feedback. It is not clear that this document should specify a usage of a mechanism that has not been fully defined. Accurate ECN may lead to different congestion responses and these will need to be defined in the CC specifications for using Accurate ECN.

WG draft 09 contained:

- Removed update to RFC 5681 - the status of the present document is Experimental, and hence this document does not update RFC 5681.

WG draft 10 contained edits following WGLC:

- Section 1.1 Implementation of new CWV: New section added to introduce the places where there are implementation flexibility.
- Section 4.4: Clarified that the MUST is to satisfy the goal to avoid a TCP sender growing a large "non-validated" cwnd, when it has not recently sent using the current size of cwnd, and fixed format of bullet 2 in 4.4.
- Section 4.5.2: rewritten section text.

WG draft 11 contained edits following IETF LC:

- Updated text in section 1.1.
- Updated text in response to AD, Gen-ART, & Sec reviews.
- LC call comments from Mirja Kuehlewind

WG draft 12 contained edits following IETF LC (Mirja Kuehlewind):

- Additional text (based on text in annexe notes) to clarify use of pipeACK rather than FlightSize.
- Corrected text on undefined pipeACK - to be consistent.
- Added text on standard TCP method (reference to RFC 5681).
- Separated text on implementation experience of "timers" into a new implementation subsection (4.5.2), to avoid this common implementation method being overlooked.

WG draft 13 contained edits following IESG Review:

- Jari/Gen-ART (note: MSS was defined)
- Kathleen Moriarty (SecDir)
- Ben Campbell
- Barry Leiba (note: reference added to section 4, rather than new wording to requirement).

11. References

11.1. Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", September 1981.

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

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

[RFC2861] Handley, M., Padhye, J., and S. Floyd, "TCP Congestion Window Validation", RFC 2861, June 2000.

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

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

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

11.2. Informative References

[All05] Allman, M. and E. Blanton, "Notes on burst mitigation for transport protocols", March 2005.

[Bis08] Biswas, I. and G. Fairhurst, "A Practical Evaluation of Congestion Window Validation Behaviour, 9th Annual Postgraduate Symposium in the Convergence of Telecommunications, Networking and Broadcasting (PGNet), Liverpool, UK", June 2008.

[Bis10] Biswas, I., Sathiaseelan, A., Secchi, R., and G. Fairhurst, "Analysing TCP for Bursty Traffic, Int'l J. of Communications, Network and System Sciences, 7(3)", June 2010.

[Bis11] Biswas, I., "PhD Thesis, Internet congestion control for variable rate TCP traffic, School of Engineering, University of Aberdeen", June 2011.

[Fai12] Sathiaseelan, A., Secchi, R., Fairhurst, G., and I. Biswas, "Enhancing TCP Performance to support Variable-Rate Traffic, 2nd Capacity Sharing Workshop, ACM CoNEXT, Nice, France, 10th December 2012.", June 2008.

[Hos15] Hossain, Z., "PhD Thesis, A Study of Mechanisms to Support Variable-rate Internet Applications over a Multi-service Satellite Platform, School of Engineering, University of Aberdeen", January 2015.

[Hug01] Hughes, A., Touch, J., and J. Heidemann, "Issues in TCP Slow-Start Restart After Idle (Work-in-Progress)", December 2001.

[Liu07] Liu, D., Allman, M., Jiny, S., and L. Wang, "Congestion Control without a Startup Phase, 5th International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet), Los Angeles, California, USA", February 2007.

[RFC7230] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, June 2014.

Authors' Addresses

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

Email: gorry@erg.abdn.ac.uk
URI: http://www.erg.abdn.ac.uk

Arjuna Sathiaseelan
University of Aberdeen
School of Engineering
Fraser Noble Building
Aberdeen, Scotland AB24 3UE
UK

Email: arjuna@erg.abdn.ac.uk
URI: http://www.erg.abdn.ac.uk

Raffaello Secchi
University of Aberdeen
School of Engineering
Fraser Noble Building
Aberdeen, Scotland AB24 3UE
UK

Email: raffaello@erg.abdn.ac.uk
URI: http://www.erg.abdn.ac.uk

TCP Maintenance and Minor Extensions (tcpm)
Internet-Draft
Intended status: Experimental
Expires: May 8, 2016

P. Hurtig
A. Brunstrom
Karlstad University
A. Petlund
Simula Research Laboratory AS
M. Welzl
University of Oslo

November 5, 2015

TCP and SCTP RTO Restart
draft-ietf-tcpm-rtorestart-10

Abstract

This document describes a modified sender-side algorithm for managing the TCP and SCTP retransmission timers that provides faster loss recovery when there is a small amount of outstanding data for a connection. The modification, RTO Restart (RTOR), allows the transport to restart its retransmission timer using a smaller timeout duration, so that the effective RTO becomes more aggressive in situations where fast retransmit cannot be used. This enables faster loss detection and recovery for connections that are short-lived or application-limited.

Status of This Memo

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

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

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

This Internet-Draft will expire on May 8, 2016.

Copyright Notice

1. Introduction

TCP and SCTP use two almost identical mechanisms to detect and recover from data loss, specified in [RFC6298][RFC5681] (for TCP) and [RFC4960] (for SCTP). First, if transmitted data is not acknowledged within a certain amount of time, a retransmission timeout (RTO) occurs, and the data is retransmitted. While the RTO is based on measured round-trip times (RTTs) between the sender and receiver, it also has a conservative lower bound of 1 second to ensure that delayed data are not mistaken as lost. Second, when a sender receives duplicate acknowledgments, or similar information via selective acknowledgments, the fast retransmit algorithm suspects data loss and can trigger a retransmission. Duplicate (and selective) acknowledgments are generated by a receiver when data arrives out-of-order. As both data loss and data reordering cause out-of-order arrival, fast retransmit waits for three out-of-order notifications before considering the corresponding data as lost. In some situations, however, the amount of outstanding data is not enough to trigger three such acknowledgments, and the sender must rely on lengthy RTOs for loss recovery.

The amount of outstanding data can be small for several reasons:

(1) The connection is limited by the congestion control when the path has a low total capacity (bandwidth-delay product) or the connection's share of the capacity is small. It is also limited by the congestion control in the first few RTTs of a connection or after an RTO when the available capacity is probed using slow-start.

(2) The connection is limited by the receiver's available buffer space.

(3) The connection is limited by the application if the available capacity of the path is not fully utilized (e.g. interactive applications), or at the end of a transfer.

While the reasons listed above are valid for any flow, the third reason is most common for applications that transmit short flows, or use a bursty transmission pattern. A typical example of applications that produce short flows are web-based applications. [RJ10] shows that 70% of all web objects, found at the top 500 sites, are too small for fast retransmit to work. [FDT13] shows that about 77% of all retransmissions sent by a major web service are sent after RTO expiry. Applications with bursty transmission patterns often send data in response to actions, or as a reaction to real life events. Typical examples of such applications are stock trading systems, remote computer operations, online games, and web-based applications using persistent connections. What is special about this class of applications is that they often are time-dependant, and extra latency can reduce the application service level [P09].

The RTO Restart (RTOR) mechanism described in this document makes the effective RTO slightly more aggressive when the amount of outstanding data is too small for fast retransmit to work, in an attempt to enable faster loss recovery while being robust to reordering. While RTOR still conforms to the requirement for when a segment can be retransmitted, specified in [RFC6298] (for TCP) and [RFC4960] (for SCTP) it could increase the risk of spurious timeouts. To determine whether this modification is safe to deploy and enable by default, further experimentation is required. Section 5 discusses experiments still needed, including evaluations in environments where the risk of spurious retransmissions are increased e.g. mobile networks with highly varying RTTs.

The remainder of this document describes RTOR and its implementation for TCP only, to make the document easier to read. However, the RTOR algorithm described in Section 4 is applicable also for SCTP. Furthermore, Section 7 details the SCTP socket API needed to control RTOR.

2. Terminology

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

This document introduces the following variables:

The number of previously unsent segments (prevunsnt): The number of segments that a sender has queued for transmission, but has not yet sent.

RTO Restart threshold (rrthresh): RTOR is enabled whenever the sum of the number of outstanding and previously unsent segments (prevunsnt) is below this threshold.

3. RTO Overview and Rationale for RTOR

The RTO management algorithm described in [RFC6298] recommends that the retransmission timer is restarted when an acknowledgment (ACK) that acknowledges new data is received and there is still outstanding data. The restart is conducted to guarantee that unacknowledged segments will be retransmitted after approximately RTO seconds. The standardized RTO timer management is illustrated in Figure 1 where a TCP sender transmits three segments to a receiver. The arrival of the first and second segment triggers a delayed ACK (delACK) [RFC1122], which restarts the RTO timer at the sender. The RTO is restarted approximately one RTT after the transmission of the third segment. Thus, if the third segment is lost, as indicated in Figure 1, the effective loss detection time become "RTO + RTT" seconds. In some situations, the effective loss detection time becomes even longer. Consider a scenario where only two segments are outstanding. If the second segment is lost, the time to expire the delACK timer will also be included in the effective loss detection time.

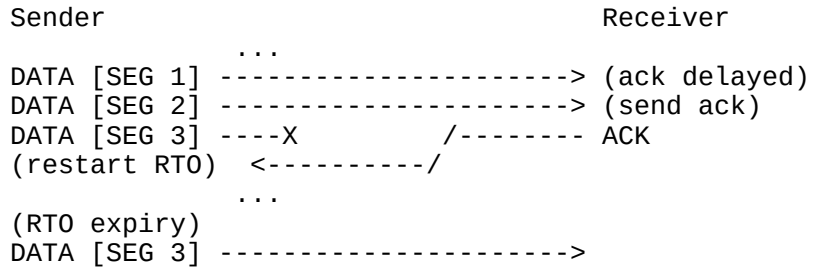


Figure 1: RTO restart example

For bulk traffic the current approach is beneficial -- it is described in [EL04] to act as a "safety margin" that compensates for some of the problems that the authors have identified with the standard RTO calculation. Notably, the authors of [EL04] also state that "this safety margin does not exist for highly interactive applications where often only a single packet is in flight". In general, however, as long as enough segments arrive at a receiver to enable fast retransmit, RTO-based loss recovery should be avoided. RTOs should only be used as a last resort, as they drastically lower the congestion window compared to fast retransmit.

Although fast retransmit is preferrable there are situations where timeouts are appropriate, or the only choice. For example, if the network is severely congested and no segments arrive RTO-based recovery should be used. In this situation, the time to recover from the loss(es) will not be the performance bottleneck. However, for connections that do not utilize enough capacity to enable fast retransmit, RTO-based loss detection is the only choice and the time required for this can become a performance bottleneck.

4. RTOR Algorithm

To enable faster loss recovery for connections that are unable to use fast retransmit, RTOR can be used. This section specifies the modifications required to use RTOR. By resetting the timer to "RTO - T_earliest", where T_earliest is the time elapsed since the earliest outstanding segment was transmitted, retransmissions will always occur after exactly RTO seconds.

This document specifies an OPTIONAL sender-only modification to TCP and SCTP which updates step 5.3 in Section 5 of [RFC6298] (and a similar update in Section 6.3.2 of [RFC4960] for SCTP). A sender that implements this method MUST follow the algorithm below:

When an ACK is received that acknowledges new data:

(1) Set T_earliest = 0.

(2) If the sum of the number of outstanding and previously unsent segments (prevunsnt) is less than an RTOR threshold (rrthresh), set T_earliest to the time elapsed since the earliest outstanding segment was sent.

(3) Restart the retransmission timer so that it will expire after (for the current value of RTO):

(a) RTO - T_earliest, if RTO - T_earliest > 0.

(b) RTO, otherwise.

The RECOMMENDED value of rrthresh is four, as this value will ensure that RTOR is only used when fast retransmit cannot be triggered. With this update, TCP implementations MUST track the time elapsed since the transmission of the earliest outstanding segment (T_earliest). As RTOR is only used when the amount of outstanding and previously unsent data is less than rrthresh segments, TCP implementations also need to track whether the amount of outstanding and previously unsent data is more, equal, or less than rrthresh segments. Although some packet-based TCP implementations (e.g.

Linux TCP) already track both the transmission times of all segments and also the number of outstanding segments, not all implementations do. Section 5.3 describes how to implement segment tracking for a general TCP implementation. To use RTOR, the calculated expiration time MUST be positive (step 3(a) in the list above); this is required to ensure that RTOR does not trigger retransmissions prematurely when previously retransmitted segments are acknowledged.

5. Discussion

Although RTOR conforms to the requirement in [RFC6298] that segments must not be retransmitted earlier than RTO seconds after their original transmission, RTOR makes the effective RTO more aggressive. In this section, we discuss the applicability and the issues related to RTOR.

5.1. Applicability

The currently standardized algorithm has been shown to add at least one RTT to the loss recovery process in TCP [LS00] and SCTP [HB11][PBP09]. For applications that have strict timing requirements (e.g. interactive web) rather than throughput requirements, using RTOR could be beneficial because the RTT and also the delACK timer of receivers are often large components of the effective loss recovery time. Measurements in [HB11] have shown that the total transfer time of a lost segment (including the original transmission time and the loss recovery time) can be reduced by 35% using RTOR. These results match those presented in [PGH06][PBP09], where RTOR is shown to significantly reduce retransmission latency.

There are also traffic types that do not benefit from RTOR. One example of such traffic is bulk transmission. The reason why bulk traffic does not benefit from RTOR is that such traffic flows mostly have four or more segments outstanding, allowing loss recovery by fast retransmit. However, there is no harm in using RTOR for such traffic as the algorithm only is active when the amount of outstanding and unsent segments are less than rrthresh (default 4).

Given that RTOR is a mostly conservative algorithm, it is suitable for experimentation as a system-wide default for TCP traffic.

5.2. Spurious Timeouts

RTOR can in some situations reduce the loss detection time and thereby increase the risk of spurious timeouts. In theory, the retransmission timer has a lower bound of 1 second [RFC6298], which limits the risk of having spurious timeouts. However, in practice most implementations use a significantly lower value. Initial

measurements show slight increases in the number of spurious timeouts when such lower values are used [RHB15]. However, further experiments, in different environments and with different types of traffic, are encouraged to quantify such increases more reliably.

Does a slightly increased risk matter? Generally, spurious timeouts have a negative effect on the network as segments are transmitted needlessly. However, recent experiments do not show a significant increase in network load for a number of realistic scenarios [RHB15]. Another problem with spurious retransmissions is related to the performance of TCP/SCTP, as the congestion window is reduced to one segment when timeouts occur [RFC5681]. This could be a potential problem for applications transmitting multiple bursts of data within a single flow, e.g. web-based HTTP/1.1 and HTTP/2.0 applications. However, results from recent experiments involving persistent web traffic [RHB15] revealed a net gain of using RTOR. Other types of flows, e.g. long-lived bulk flows, are not affected as the algorithm is only applied when the amount of outstanding and unsent segments is less than rrthresh. Furthermore, short-lived and application-limited flows are typically not affected as they are too short to experience the effect of congestion control or have a transmission rate that is quickly attainable.

While a slight increase in spurious timeouts has been observed using RTOR, it is not clear whether the effects of this increase mandate any future algorithmic changes or not -- especially since most modern operating systems already include mechanisms to detect [RFC3522][RFC3708][RFC5682] and resolve [RFC4015] possible problems with spurious retransmissions. Further experimentation is needed to determine this and thereby move this specification from experimental to the standards track. For instance, RTOR has not been evaluated in the context of mobile networks. Mobile networks often incur highly variable RTTs (delay spikes), due to e.g. handovers, and would therefore be a useful scenario for further experimentation.

5.3. Tracking Outstanding and Previously Unsent Segments

The method of tracking outstanding and previously unsent segments will probably differ depending on the actual TCP implementation. For packet-based TCP implementations, tracking outstanding segments is often straightforward and can be implemented using a simple counter. For byte-based TCP stacks it is a more complex task. Section 3.2 of [RFC5827] outlines a general method of tracking the number of outstanding segments. The same method can be used for RTOR. The implementation will have to track segment boundaries to form an understanding as to how many actual segments have been transmitted, but not acknowledged. This can be done by the sender tracking the boundaries of the rrthresh segments on the right side of the current

window (which involves tracking rrthresh + 1 sequence numbers in TCP). This could be done by keeping a circular list of the segment boundaries, for instance. Cumulative ACKs that do not fall within this region indicate that at least rrthresh segments are outstanding, and therefore RTOR is not enabled. When the outstanding window becomes small enough that RTOR can be invoked, a full understanding of the number of outstanding segments will be available from the rrthresh + 1 sequence numbers retained. (Note: the implicit sequence number consumed by the TCP FIN bit can also be included in the tracking of segment boundaries.)

Tracking the number of previously unsent segments depends on the segmentation strategy used by the TCP implementation, not whether it is packet-based or byte-based. In the case segments are formed directly on socket writes, the process of determining the number of previously unsent segments should be trivial. In the case that unsent data can be segmented (or re-segmented) as long as it still is unsent, a straightforward strategy could be to divide the amount of unsent data (in bytes) with the SMSS to obtain an estimate. In some cases, such an estimation could be too simplistic, depending on the segmentation strategy of the TCP implementation. However, this estimation is not critical to RTOR. The tracking of prevunsnt is only made to optimize a corner case in which RTOR was unnecessarily disabled. Implementations can use a simplified method by setting prevunsnt to rrthresh whenever previously unsent data is available, and set prevunsnt to zero when no new data is available. This will disable RTOR in the presence of unsent data and only use the number of outstanding segments to enable/disable RTOR.

6. Related Work

There are several proposals that address the problem of not having enough ACKs for loss recovery. In what follows, we explain why the mechanism described here is complementary to these approaches:

The limited transmit mechanism [RFC3042] allows a TCP sender to transmit a previously unsent segment for each of the first two dupACKs. By transmitting new segments, the sender attempts to generate additional dupACKs to enable fast retransmit. However, limited transmit does not help if no previously unsent data is ready for transmission. [RFC5827] specifies an early retransmit algorithm to enable fast loss recovery in such situations. By dynamically lowering the number of dupACKs needed for fast retransmit (dupthresh), based on the number of outstanding segments, a smaller number of dupACKs is needed to trigger a retransmission. In some situations, however, the algorithm is of no use or might not work properly. First, if a single segment is outstanding, and lost, it is impossible to use early retransmit. Second, if ACKs are lost, early

retransmit cannot help. Third, if the network path reorders segments, the algorithm might cause more spurious retransmissions than fast retransmit. The recommended value of RTOR's rrthresh variable is based on the dupthresh, but is possible to adapt to allow tighter integration with other experimental algorithms such as early retransmit.

Tail Loss Probe [TLP] is a proposal to send up to two "probe segments" when a timer fires which is set to a value smaller than the RTO. A "probe segment" is a new segment if new data is available, else a retransmission. The intention is to compensate for sluggish RTO behavior in situations where the RTO greatly exceeds the RTT, which, according to measurements reported in [TLP], is not uncommon. Furthermore, TLP also tries to circumvent the congestion window reset to one segment by instead enabling fast recovery. The Probe timeout (PTO) is normally two RTTs, and a spurious PTO is less risky than a spurious RTO because it would not have the same negative effects (clearing the scoreboard and restarting with slow-start). TLP is a more advanced mechanism than RTOR, requiring e.g. SACK to work, and is often able to reduce loss recovery times more. However, it also increases the amount of spurious retransmissions noticeably, as compared to RTOR [RHB15].

TLP is applicable in situations where RTOR does not apply, and it could overrule (yielding a similar general behavior, but with a lower timeout) RTOR in cases where the number of outstanding segments is smaller than four and no new segments are available for transmission. The PTO has the same inherent problem of restarting the timer on an incoming ACK, and could be combined with a strategy similar to RTOR's to offer more consistent timeouts.

7. SCTP Socket API Considerations

This section describes how the socket API for SCTP defined in [RFC6458] is extended to control the usage of RTO restart for SCTP.

Please note that this section is informational only.

7.1. Data Types

This section uses data types from [IEEE.1003-1G.1997]: uintN_t means an unsigned integer of exactly N bits (e.g., uint16_t). This is the same as in [RFC6458].

7.2. Socket Option for Controlling the RTO Restart Support (SCTP_RTO_RESTART)

This socket option allows the enabling or disabling of RTO Restart for SCTP associations.

Whether RTO Restart is enabled or not per default is implementation specific.

This socket option uses IPPROTO_SCTP as its level and SCTP_RTO_RESTART as its name. It can be used with getsockopt() and setsockopt(). The socket option value uses the following structure defined in [RFC6458]:

```
struct sctp_assoc_value {
  sctp_assoc_t assoc_id;
  uint32_t assoc_value;
};
```

assoc_id: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets, this parameter indicates upon which association the user is performing an action. The special sctp_assoc_t SCTP_{FUTURE|CURRENT|ALL}_ASSOC can also be used in assoc_id for setsockopt(). For getsockopt(), the special value SCTP_FUTURE_ASSOC can be used in assoc_id, but it is an error to use SCTP_{CURRENT|ALL}_ASSOC in assoc_id.

assoc_value: A non-zero value encodes the enabling of RTO restart whereas a value of 0 encodes the disabling of RTO restart.

sctp_opt_info() needs to be extended to support SCTP_RTO_RESTART.

8. IANA Considerations

This memo includes no request to IANA.

9. Security Considerations

This document specifies an experimental sender-only modification to TCP and SCTP. The modification introduces a change in how to set the retransmission timer’s value when restarted. Therefore, the security considerations found in [RFC6298] apply to this document. No additional security problems have been identified with RTO Restart at this time.

10. Acknowledgements

The authors wish to thank Michael Tuexen for contributing the SCTP Socket API considerations and Godred Fairhurst, Yuchung Cheng, Mark Allman, Anantha Ramaiah, Richard Scheffenegger, Nicolas Kuhn, Alexander Zimmermann, and Michael Scharf for commenting on the draft and the ideas behind it.

All the authors are supported by RITE (http://riteproject.eu/), a research project (ICT-317700) funded by the European Community under its Seventh Framework Program. The views expressed here are those of the author(s) only. The European Commission is not liable for any use that may be made of the information in this document.

11. Changes from Previous Versions

RFC-Editor note: please remove this section prior to publication.

11.1. Changed from draft-ietf-...-09 to -10

- Changed wording in abstract, from "delay" to "timeout duration".

11.2. Changed from draft-ietf-...-08 to -09

- Clarified, in the abstract, that the modified restart causes a smaller retransmission delay in total.
- Clarified, in the introduction, that the fast retransmit algorithm may cause retransmissions upon receiving duplicate acknowledgments, not that it unconditionally does so.
- Changed wording from "to proposed standard" to "to the standards track".
- Changed algorithm description so that a TCP sender MUST track the time elapsed since the transmission of the earliest outstanding segment. This was not explicitly stated in previous versions of the draft.

11.3. Changes from draft-ietf-...-07 to -08

- Clarified, at multiple places in the document, that the modification only causes the effective RTO to be more aggressive, not the actual RTO.
- Removed information in the introduction that was too detailed, i.e., material that is hard to understand without knowing details of the algorithm.

- Changed the name of Section 3 to more correctly capture the actual contents of the section.
- Re-arranged the text in Section 3 to have a more logical structure.
- Moved text from the algorithm description (Section 4) to the introduction of the discussion section (Section 5). The text was discussing the possible effects of the algorithm more than describing the actual algorithm.
- Clarified why the RECOMMENDED value of rrthresh is four.
- Reworked the introduction to be suitable for both TCP and SCTP.

11.4. Changes from draft-ietf-...-06 to -07

- Clarified, at multiple places in the document, that the modification is sender-only.
- Added an explanation (in the introduction) to why the mechanism is experimental and what experiments are missing.
- Added a sentence in Section 4 to clarify that the section is the one describing the actual modification.

11.5. Changes from draft-ietf-...-05 to -06

- Added socket API considerations, after discussing the draft in tsvwg.

11.6. Changes from draft-ietf-...-04 to -05

- Introduced variable to track the number of previously unsent segments.
- Clarified many concepts, e.g. extended the description on how to track outstanding and previously unsent segments.
- Added a reference to initial measurements on the effects of using RTOR.
- Improved wording throughout the document.

11.7. Changes from draft-ietf-...-03 to -04

- Changed the algorithm to allow RTOR when there is unsent data available, but the cwnd does not allow transmission.
- Changed the algorithm to not trigger if RTOR <= 0.
- Made minor adjustments throughout the document to adjust for the algorithmic change.
- Improved the wording throughout the document.

11.8. Changes from draft-ietf-...-02 to -03

- Updated the document to use "RTOR" instead of "RTO Restart" when refering to the modified algorithm.
- Moved document terminology to a section of its own.
- Introduced the rrthresh variable in the terminology section.
- Added a section to generalize the tracking of outstanding segments.
- Updated the algorithm to work when the number of outstanding segments is less than four and one segment is ready for transmission, by restarting the timer when new data has been sent.
- Clarified the relationship between fast retransmit and RTOR.
- Improved the wording throughout the document.

11.9. Changes from draft-ietf-...-01 to -02

- Changed the algorithm description in Section 3 to use formal RFC 2119 language.
- Changed last paragraph of Section 3 to clarify why the RTO restart algorithm is active when less than four segments are outstanding.
- Added two paragraphs in Section 4.1 to clarify why the algorithm can be turned on for all TCP traffic without having any negative effects on traffic patterns that do not benefit from a modified timer restart.
- Improved the wording throughout the document.
- Replaced and updated some references.

11.10. Changes from draft-ietf-...-00 to -01

- Improved the wording throughout the document.
- Removed the possibility for a connection limited by the receiver's advertised window to use RTO restart, decreasing the risk of spurious retransmission timeouts.
- Added a section that discusses the applicability of and problems related to the RTO restart mechanism.
- Updated the text describing the relationship to TLP to reflect updates made in this draft.
- Added acknowledgments.

12. References

12.1. Normative References

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <http://www.rfc-editor.org/info/rfc1122>.

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

[RFC3042] Allman, M., Balakrishnan, H., and S. Floyd, "Enhancing TCP's Loss Recovery Using Limited Transmit", RFC 3042, DOI 10.17487/RFC3042, January 2001, <http://www.rfc-editor.org/info/rfc3042>.

[RFC3522] Ludwig, R. and M. Meyer, "The Eifel Detection Algorithm for TCP", RFC 3522, DOI 10.17487/RFC3522, April 2003, <http://www.rfc-editor.org/info/rfc3522>.

[RFC3708] Blanton, E. and M. Allman, "Using TCP Duplicate Selective Acknowledgement (DSACKs) and Stream Control Transmission Protocol (SCTP) Duplicate Transmission Sequence Numbers (TSNs) to Detect Spurious Retransmissions", RFC 3708, DOI 10.17487/RFC3708, February 2004, <http://www.rfc-editor.org/info/rfc3708>.

[RFC4015] Ludwig, R. and A. Gurtov, "The Eifel Response Algorithm for TCP", RFC 4015, DOI 10.17487/RFC4015, February 2005, <http://www.rfc-editor.org/info/rfc4015>.

[RFC4960] Stewart, R., Ed., "Stream Control Transmission Protocol", RFC 4960, DOI 10.17487/RFC4960, September 2007, <http://www.rfc-editor.org/info/rfc4960>.

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

[RFC5682] Sarolahti, P., Kojo, M., Yamamoto, K., and M. Hata, "Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP", RFC 5682, DOI 10.17487/RFC5682, September 2009, <http://www.rfc-editor.org/info/rfc5682>.

[RFC5827] Allman, M., Avrachenkov, K., Ayesta, U., Blanton, J., and P. Hurtig, "Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)", RFC 5827, DOI 10.17487/RFC5827, May 2010, <http://www.rfc-editor.org/info/rfc5827>.

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

12.2. Informative References

[EL04] Ekstroem, H. and R. Ludwig, "The Peak-Hopper: A New End-to-End Retransmission Timer for Reliable Unicast Transport", IEEE INFOCOM 2004, March 2004.

[FDT13] Flach, T., Dukkipati, N., Terzis, A., Raghavan, B., Cardwell, N., Cheng, Y., Jain, A., Hao, S., Katz-Bassett, E., and R. Govindan, "Reducing Web Latency: the Virtue of Gentle Aggression", Proc. ACM SIGCOMM Conf., August 2013.

[HB11] Hurtig, P. and A. Brunstrom, "SCTP: designed for timely message delivery?", Springer Telecommunication Systems 47 (3-4), August 2011.

[IEEE.1003-1G.1997] Institute of Electrical and Electronics Engineers, "Protocol Independent Interfaces", IEEE Standard 1003.1G, March 1997.

[LS00] Ludwig, R. and K. Sklower, "The Eifel retransmission timer", ACM SIGCOMM Comput. Commun. Rev., 30(3), July 2000.

[P09] Petlund, A., "Improving latency for interactive, thin-stream applications over reliable transport", Unipub PhD Thesis, Oct 2009.

[PBP09] Petlund, A., Beskow, P., Pedersen, J., Paaby, E., Griwodz, C., and P. Halvorsen, "Improving SCTP Retransmission Delays for Time-Dependent Thin Streams", Springer Multimedia Tools and Applications, 45(1-3), 2009.

[PGH06] Pedersen, J., Griwodz, C., and P. Halvorsen, "Considerations of SCTP Retransmission Delays for Thin Streams", IEEE LCN 2006, November 2006.

[RFC6458] Stewart, R., Tuexen, M., Poon, K., Lei, P., and V. Yasevich, "Sockets API Extensions for the Stream Control Transmission Protocol (SCTP)", RFC 6458, DOI 10.17487/RFC6458, December 2011, <http://www.rfc-editor.org/info/rfc6458>.

[RHB15] Rajiullah, M., Hurtig, P., Brunstrom, A., Petlund, A., and M. Welzl, "An Evaluation of Tail Loss Recovery Mechanisms for TCP", ACM SIGCOMM CCR 45 (1), January 2015.

[RJ10] Ramachandran, S., "Web metrics: Size and number of resources", Google http://code.google.com/speed/articles/web-metrics.html, May 2010.

[TLP] Dukkipati, N., Cardwell, N., Cheng, Y., and M. Mathis, "TCP Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses", Internet-draft draft-dukkipati-tcpm-tcp-loss-probe-01.txt, February 2013.

Authors' Addresses

Per Hurtig
Karlstad University
Universitetsgatan 2
Karlstad 651 88
Sweden

Phone: +46 54 700 23 35
Email: per.hurtig@kau.se

Anna Brunstrom
Karlstad University
Universitetsgatan 2
Karlstad 651 88
Sweden

Phone: +46 54 700 17 95
Email: anna.brunstrom@kau.se

Andreas Petlund
Simula Research Laboratory AS
P.O. Box 134
Lysaker 1325
Norway

Phone: +47 67 82 82 00
Email: apetlund@simula.no

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

Phone: +47 22 85 24 20
Email: michawe@ifi.uio.no

TCPM WG
Internet Draft
Updates: 9293
Intended status: Standards Track
Expires: April 2023

J. Touch
Independent Consultant
Wes Eddy
MTI Systems

October 22, 2022

TCP Extended Data Offset Option
draft-ietf-tcpm-tcp-edo-13.txt

Abstract

TCP segments include a Data Offset field to indicate space for TCP options but the size of the field can limit the space available for complex options such as SACK and Multipath TCP and can limit the combination of such options supported in a single connection. This document updates RFC 9293 with an optional TCP extension to that space to support the use of multiple large options. It also explains why the initial SYN of a connection cannot be extending a single segment.

Table of Contents

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

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

1. Introduction

TCP's Data Offset (DO)is a 4-bit field, which indicates the number of 32-bit words of the entire TCP header [RFC9293]. This limits the current total header size to 60 bytes, of which the basic header occupies 20, leaving 40 bytes for options. These 40 bytes are increasingly becoming a limitation to the development of advanced capabilities, such as when SACK [RFC2018][RFC6675] is combined with either Multipath TCP [RFC8684], TCP-AO [RFC5925], or TCP Fast Open [RFC7413].

This document specifies the TCP Extended Data Offset (EDO) option, and is independent of (and thus compatible with) IPv4 and IPv6. EDO extends the space available for TCP options, except for the initial SYN and SYN/ACK. This document also explains why the option space of the initial SYN segments cannot be extended as individual segments without severe impact on TCP's initial handshake and the SYN/ACK limitation that results from potential middlebox misbehavior. Multiple other TCP extensions are being considered in the TCPM working group in order to address the case of SYN and SYN/ACK segments [Bo14][Br14][To22]. Some of these other extensions can work in conjunction with EDO (e.g., [To22]).

2. Conventions used in this document

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

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

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

3. Motivation

TCP supports headers with a total length of up to 15 32-bit words, as indicated in the 4-bit Data Offset field [RFC9293]. This accounts

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

TCP connections already use this option space for a variety of capabilities. These include Maximum Segment Size (MSS) [RFC9293], Window Scale (WS) [RFC7323], Timestamp (TS) [RFC7323], Selective Acknowledgement (SACK) [RFC2018][RFC6675], TCP Authentication Option (TCP-AO) [RFC5925], Multipath TCP (MP-TCP)_[RFC8684], and TCP User Timeout [RFC5482]. Some options occur only in a SYN or SYN/ACK (MSS, WS), and others vary in size when used in SYN vs. non-SYN segments.

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

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

4. Requirements for Extending TCP's Data Offset

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

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

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

5. The TCP EDO Option

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

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

5.1. EDO Supported

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

Figure 1 TCP EDO Supported option

An endpoint seeking to enable EDO includes the EDO Supported option in the initial SYN. If receiver of that SYN agrees to use EDO, it responds with the EDO Supported option in the SYN/ACK. The EDO Supported option does not extend the TCP option space.

>> Connections using EDO MUST negotiate its availability during the SYN exchange of the initial three-way handshake.

>> An endpoint confirming and agreeing to EDO use MUST respond with the EDO Supported option in its SYN/ACK.

The SYN/ACK uses only the EDO Supported option (and not the EDO Extension option, below) because it may not yet be safe to extend the option space in the reverse direction due to potential middlebox misbehavior (see Section 7.2). Extension of the SYN and SYN/ACK space is addressed as a separate option (see Section 8.7).

5.2. EDO Extension

When EDO is successfully negotiated, all other segments use the EDO Extension option, of which there are two variants (Figure 2 and Figure 3). Both variants are considered equivalent and either variant can be used in any segment where the EDO Extension option is required. Both variants add a Header_Length field (in network-standard byte order), indicating the length of the entire TCP header in 32-bit words. Figure 3 depicts the longer variant, which includes an additional Segment_Length field, which is identical to the TCP

pseudoheader TCP Length field and used to detect when segments have been altered in ways that would interfere with EDO (discussed further in Section 5.3).

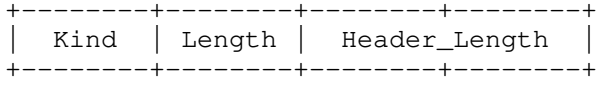


Figure 2 TCP EDO Extension option - simple variant

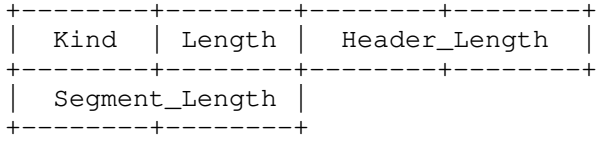


Figure 3 TCP EDO Extension option - with segment length verification

>> Once enabled on a connection, all segments in both directions MUST include the EDO Extension option. Segments not needing extension MUST set the EDO Extension option Header Length field equal to the Data Offset length.

>> The EDO Extension option MAY be used only if confirmed when the connection transitions to the ESTABLISHED state, e.g., a client is enabled after receiving the EDO Supported option in the SYN/ACK and the server is enabled after seeing the EDO Extension option in the final ACK of the three-way handshake. If either of those segments lacks the appropriate EDO option, the connection MUST NOT use any EDO options on any other segments.

Internet paths may vary after connection establishment, introducing misbehaving middleboxes (see Section 7.2). Using EDO on all segments in both directions allows this condition to be detected.

>> The EDO Supported option MAY occur in an initial SYN as desired (e.g., as expressed by the user/application) and in the SYN/ACK as confirmation, but MUST NOT be inserted in other segments. If the EDO Supported option is received in other segments, it MUST be silently ignored.

>> If EDO has not been negotiated and agreed, the EDO Extension option MUST be silently ignored on subsequent segments. The EDO Extension option MUST NOT be sent in an initial SYN segment or SYN/ACK, and MUST be silently ignored and not acknowledged if so received.

>> If EDO has been negotiated, any subsequent segments arriving without the EDO Extension option MUST be silently ignored. Such events MAY be logged as warning errors and logging MUST be rate limited.

When processing a segment, EDO needs to be visible within the area indicated by the Data Offset field, so that processing can use the EDO Header_length to override the field for that segment.

>> The EDO Extension option MUST occur within the space indicated by the TCP Data Offset.

>> The EDO Extension option indicates the total length of the header. The EDO Header_length field MUST NOT exceed that of the total segment size (i.e., TCP Length).

>> The EDO Header Length MUST be at least as large as the TCP Data Offset field of the segment in which they both appear. When the EDO Header Length equals the Data Offset length, the EDO Extension option is present but it does not extend the option space. When the EDO Header Length is invalid, the TCP segment MUST be silently dropped.

>> The EDO Supported option SHOULD be aligned on a 16-bit boundary and the EDO Extension option SHOULD be aligned on a 32-bit boundary, in both cases for simpler processing.

For example, a segment with only EDO would have a Data Offset of 6 or 7 (depending on the EDO Extension variant used), where EDO would be the first option processed, at which point the EDO Extension option would override the Data Offset and processing would continue until the end of the TCP header as indicated by the EDO Header_length field.

There are cases where it might be useful to process other options before EDO, notably those that determine whether the TCP header is valid, such as authentication, encryption, or alternate checksums. In those cases, the EDO Extension option is preferably the first option after a validation option, and the payload after the Data Offset is treated as user data for the purposes of validation.

>> The EDO Extension option SHOULD occur as early as possible, either first or just after any authentication or encryption, and SHOULD be the last option covered by the Data Offset value.

Other options are generally handled in the same manner as when the EDO option is not active, unless they interact with other options.

One such example is TCP-AO [RFC5925], which optionally ignores the contents of TCP options, so it would need to be aware of EDO to operate correctly when options are excluded from the HMAC calculation.

>> Options that depend on other options, such as TCP-AO [RFC5925] (which may include or exclude options in MAC calculations) MUST also be augmented to interpret the EDO Extension option to operate correctly.

5.3. The two EDO Extension variants

There are two variants of the EDO Extension option; one includes a copy of the TCP segment length, copied from the TCP pseduoheader [RFC9293]. The Segment_Length field is added to the longer variant to detect when segments are incorrectly and inappropriately merged by middleboxes or TCP offload processing but without consideration for the additional option space indicated by the EDO Header_Length field. Such effects are described in further detail in Section 7.2.

>> An endpoint MAY use either variant of the EDO Extension option interchangeably.

When the longer, 6-byte variant is used, the Segment_Length field is used to check whether modification of the segment was performed consistent with knowledge of the EDO option. The Segment_Length field will detect any modification of the length of the segment, such as might occur when segments are split or merged, that occurs without also updating the Segment Length field as well. The Segment Length field thus helps endpoints detects devices that merge or split TCP segments without support for EDO. Devices that merge or split TCP segments that support EDO would update the Segment Length field as needed but would also ensure that the user data is handled separately from the extended option space indicate by EDO.

>> When an endpoint creates a new segment using the 6-byte EDO Extension option, the Segment_Length field is initialized with a copy of the segment length from the TCP pseudoheader.

>> When an endpoint receives a segment using the 6-byte EDO Extension option, it MUST validate the Segment_Length field with the length of the segment as indicated in the TCP pseudoheader. If the segment lengths do not match, the segment MUST be discarded and an error SHOULD be logged in a rate-limited manner.

>> The 6-byte EDO Extension variant SHOULD be used where middlebox or TCP offload support could merge or split TCP segments without

consideration for the EDO option. Because these conditions could occur at either endpoint or along the network path, the 6-byte variant SHOULD be preferred until sufficient evidence for safe use of the 4-byte variant is determined by the community.

The field will not detect other modification of the TCP user data; such modifications would need more complex detection mechanisms, such as checksums or hashes. When these are used, as with IPsec or TCP-AO, the 4-byte variant is sufficient.

>> The 4-byte EDO Extension variant is sufficient when EDO is used in conjunction with other mechanisms that provide integrity protection, such as IPsec or TCP-AO.

6. TCP EDO Interaction with TCP

The following subsections describe how EDO interacts with the TCP specification [RFC9293].

6.1. TCP User Interface

The TCP EDO option is enabled on a connection using a mechanism similar to any other per-connection option. In Unix systems, this is typically performed using the 'setsockopt' system call.

>> Implementations can also employ system-wide defaults, however systems SHOULD NOT activate this extension by default to avoid interfering with legacy applications.

>> Due to the potential impacts of legacy middleboxes (discussed in Section 7), a TCP implementation supporting EDO SHOULD log any events within an EDO connection when options that are malformed or show other evidence of tampering arrive. An operating system MAY choose to cache the list of destination endpoints where this has occurred with and block use of EDO on future connections to those endpoints, but this cache MUST be accessible to users/applications on the host. Note that such endpoint assumptions can vary in the presence of load balancers where server implementations vary behind such balancers.

6.2. TCP States and Transitions

TCP EDO does not alter the existing TCP state or state transition mechanisms.

6.3. TCP Segment Processing

TCP EDO alters segment processing during the TCP option processing step. Once detected, the TCP EDO Extension option overrides the TCP Data Offset field for all subsequent option processing. Option processing continues at the next option (if present) after the EDO Extension option.

6.4. Impact on TCP Header Size

The TCP EDO Supported option increases SYN header length by a minimum of 2 bytes but could increase it by more depending on 32-bit word alignment. Currently popular SYN options total 19 bytes, which leaves more than enough room for the EDO Supported option:

- SACK permitted (2 bytes in SYN, optionally 2 + 8N bytes after) [RFC2018][RFC6675]
- Timestamp (10 bytes) [RFC7323]
- Window scale (3 bytes) [RFC7323]
- MSS option (4 bytes) [RFC9293]

Adding the EDO Supported option would result in a total of 21 bytes of SYN option space.

Subsequent segments would use 10 bytes of option space without any SACK blocks (TS only; WS and MSS are used only in SYN and SYN/ACK) or allow up to 3 SACK blocks before needing to use EDO; with EDO, the number of SACK blocks or additional options would be substantially increased. There are also other options that are emerging in the SYN, including TCP Fast Open, which uses another 6-18 (typically 10) bytes in the SYN/ACK of the first connection and in the SYN of subsequent connections [RFC7413].

TCP EDO can also be negotiated in SYNs with either of the following large options:

- TCP-AO (authentication) (16 bytes) [RFC5925]
- Multipath TCP (12 bytes in SYN and SYN/ACK, 20 after) [RFC8684]

Including TCP-AO with TS, WS, SACK increases the SYN option space use to 35 bytes; with Multipath TCP the use is 31 bytes. When Multipath TCP is enabled with the typical options, later segments would require 30 bytes without SACK, thus limiting the SACK option

to one block unless EDO is also supported on at least non-SYN segments.

The full combination of the above options (47 bytes for TS, WS, MSS, SACK, TCP-AO, and MPTCP) does not fit in the existing SYN option space and (as noted) that space cannot be extended within a single SYN segment. There has been a proposal to change TS to a 2 byte "TS permitted" signal in the initial SYN, provided it can be safely enabled during the connection later or might be avoided completely [Ni15]. Even using "TS-permitted", the total space is still too large to support in the initial SYN without SYN option space extension [Bo14][Br14][To22].

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

6.5. Connectionless Resets

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

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

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

6.6. ICMP Handling

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

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

7. Interactions with Middleboxes

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

There are several cases to consider:

- Typical NAT/NAPT devices, which modify only IP address and/or TCP port number fields (with associated TCP checksum updates)

- Middleboxes that try to reconstitute TCP data streams, such as for deep-packet inspection for virus scanning

- Middleboxes that modify known TCP header fields

- Middleboxes that rewrite TCP segments

7.1. Middlebox Coexistence with EDO

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

NATs and NAPTs, which rewrite IP address and/or transport port fields, are the most common form of middlebox and are not affected by the EDO option.

Middleboxes that support EDO would be those that correctly parse the EDO option. Such boxes can reconstitute the TCP data stream correctly or can modify header fields and/or rewrite segments without impact to EDO.

Conventional TCP proxies terminate the TCP connection in both directions and thus operate as TCP endpoints, such as when a client-middlebox and middlebox-server each have separate TCP connections. They would support EDO by following the host requirements herein on both connections. The use of EDO on one connection is independent of its use on the other in this case.

7.2. Middlebox Interference with EDO

Middleboxes that do not support EDO cannot coexist with its use when they modify segment boundaries or do not forward unknown (e.g., the EDO) options.

So-called "transparent" rewriting proxies, which inappropriately and incorrectly modify TCP segment boundaries, might mix option information with user data if they did not support EDO. Such devices might also interfere with other TCP options such as TCP-AO. There are three types of such boxes:

- Those that process received options and transmit sent options separately, i.e., although they rewrite segments, they behave as TCP endpoints in both directions.

- Those that split segments, taking a received segment and emitting two or more segments with revised headers.

- Those that join segments, receiving multiple segments and emitting a single segment whose data is the concatenation of the components.

In all three cases, EDO is either treated as independent on different sides of such boxes or not. If independent, EDO would either be correctly terminated in either or both directions or disabled due to lack of SYN/ACK confirmation in either or both directions. Problems would occur only when TCP segments with EDO are combined or split while ignoring the EDO option. In the split case, the key concern is if the split happens within the option extension space or if EDO is silently copied to both segments without copying the corresponding extended option space contents. However, the most comprehensive study of these cases indicates that "although middleboxes do split and coalesce segments, none did so while passing unknown options" [Ho11].

Note that the second and third types of middlebox behaviors listed above may create syndromes similar to TCP transmit and receive hardware offload engines that incorrectly modify segments with unknown options.

Middleboxes that silently remove options that they do not implement have been observed [Ho11]. Such boxes interfere with the use of the EDO Extension option in the SYN and SYN/ACK segments because extended option space would be misinterpreted as user data if the EDO Extension option were removed, and this cannot be avoided. This is one reason that SYN and SYN/ACK extension requires alternate

mechanisms (see Section 8.7). It is also the reason for the 6-byte EDO Extension variant (see Section 5.3), which can detect such merging or splitting of segments. Further, if such middleboxes become present on a path they could cause similar misinterpretation on segments exchanged in the ESTABLISHED and subsequent states. As a result, this document requires that the EDO Extension option be avoided on the SYN/ACK and that this option needs to be used on all segments once successfully negotiated and encourages use of the 6-byte EDO Extension variant.

Deep-packet inspection systems that inspect TCP segment payloads or attempt to reconstitute the data stream would incorrectly include option data in the reconstituted user data stream, which might interfere with their operation.

>> It can be important to detect misbehavior that could cause EDO space to be misinterpreted as user data. In such cases, EDO SHOULD be used in conjunction with an integrity protection mechanism. This includes the 6-byte EDO Extension variant or stronger mechanisms such as IPsec, TCP-AO, etc. It is useful to note that such protection only helps non-compliant components and enable avoidance (e.g., disabling EDO), but integrity protection alone cannot correct the misinterpretation of EDO space as user data.

This situation is similar to that of ECN and ICMP support in the Internet. In both cases, endpoints have evolved mechanisms for detecting and robustly operating around "black holes". Very similar algorithms are expected to be applicable for EDO.

8. Comparison to Previous Proposals

EDO is the latest in a long line of attempts to increase TCP option space [Al06][Ed08][Ko04][Ra12][Yo11]. The following is a comparison of these approaches to EDO, based partly on a previous summary [Ra12]. This comparison differs from that summary by using a different set of success criteria.

8.1. EDO Criteria

Our criteria for a successful solution are as follows:

- Zero-cost fallback to legacy endpoints.
- Minimal impact on middlebox compatibility.
- No additional side-effects.

Zero-cost fallback requires that upgraded hosts incur no penalty for attempting to use EDO. This disqualifies dual-stack approaches, because the client might have to delay connection establishment to wait for the preferred connection mode to complete. Note that the impact of legacy endpoints that silently reflect unknown options are not considered, as they are already non-compliant with existing TCP requirements [RFC9293].

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

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

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

8.2. Summary of Approaches

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

1. Use of a TCP option.

2. Redefinition of the existing TCP header fields.

3. Use of option space in multiple TCP segments (split across multiple segments).

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

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

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

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

8.3. Extended Segments

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

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

8.4. TCPx2

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

8.5. LO/SLO

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

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

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

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

8.6. LOIC

TCP Long Options by Invalid Checksum is a dual-stack approach that uses two initial SYNS to initiate all updated connections [Yo11]. One SYN negotiates the new option and the other SYN payload contains only the entire options. The negotiation SYN is compliant with existing procedures, but the option SYN has a deliberately incorrect TCP checksum (decremented by 2). A legacy endpoint would discard the segment with the incorrect checksum and respond to the negotiation SYN without the LO option.

Use of the option SYN and its incorrect checksum both interfere with other legacy components. Segments with incorrect checksums will be silently dropped by most middleboxes, including NATs/NAPTs. Use of two SYNs creates side-effects that can delay connections to upgraded endpoints, notably when the option SYN is lost or the SYNs arrive out of order. Finally, by not allowing other options in the negotiation SYN, all connections to legacy endpoints either use no

options or require a separate connection attempt (either concurrent or subsequent).

8.7. Problems with Extending the Initial SYN

The key difficulty with most previous proposals is the desire to extend the option space in all TCP segments, including the initial SYN, i.e., SYN with no ACK, typically the first segment of a connection, as well as possibly the SYN/ACK. It has proven difficult to extend space within the segment of the initial SYN in the absence of prior negotiation while maintaining current TCP three-way handshake properties, and it may be similarly challenging to extend the SYN/ACK (depending on asymmetric middlebox assumptions).

A new TCP option cannot extend the Data Offset of a single TCP initial SYN segment and cannot extend a SYN/ACK in a single segment when considering misbehaving middleboxes. All TCP segments, including the initial SYN and SYN/ACK, may include user data in the payload data [RFC9293], and this can be useful for some proposed features such as TCP Fast Open [RFC7413]. Legacy endpoints that ignore the new option would process the payload contents as user data and send an ACK. Once ACK'd, this data cannot be removed from the user stream.

The Reserved TCP header bits cannot be redefined easily, even though three of the six total bits have already been redefined (ECE/CWR [RFC3168] and NS [RFC3540]). Legacy endpoints have been known to reflect received values in these fields; this was safely dealt with for ECN but would be difficult here [RFC3168].

TCP initial SYN (SYN and not ACK) segments can use every other TCP header field except the Acknowledgement number, which is not used because the ACK field is not set. In all other segments, all fields except the three remaining Reserved header bits are actively used. The total amount of available header fields, in either case, is insufficient to be useful in extending the option space.

The representation of TCP options can be optimized to minimize the space needed. In such cases, multiple Kind and Length fields are combined, so that a new Kind would indicate a specific combination of options, whose order is fixed and whose length is indicated by one Length field. Most TCP options use fields whose size is much larger than the required Kind and Length components, so the resulting efficiency is typically insufficient for additional options.

The option space of an initial SYN segment might be extended by using multiple initial segments (e.g., multiple SYNs or a SYN and non-SYN) or based on the context of previous or parallel connections. This method may also be needed to extend space in the SYN/ACK in the presence of misbehaving middleboxes. Because of their potential complexity, these approaches are addressed in separate documents [Bo14][Br14][To22].

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

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

9. Implementation Issues

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

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

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

Some TCP segment offload devices (TSOs) incorrectly attempt to process segments with unknown options. EDO data needs to be passed to the protocol stack as part of the option space, not integrated with the user segment, to allow the offload to independently determine user data segment boundaries and combine them correctly with the extended option data. Issues with incorrect resegmentation

by an offload engine can be detected in the same way as middlebox tampering.

10. Security Considerations

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

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

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

11. IANA Considerations

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

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

12. References

12.1. Normative References

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

[RFC9293] Eddy, W. (Ed.), "Transmission Control Protocol", STD 7, RFC 9293, Aug. 2022.

12.2. Informative References

[Al06] Allman, M., "TCPx2: Don’t Fence Me In", draft-allman-tcpx2-hack-00 (work in progress), May 2006.

[Bo14] Borman, D., "TCP Four-Way Handshake", draft-borman-tcp4way-00 (work in progress), October 2014.

[Br14] Briscoe, B., "Inner Space for TCP Options", draft-briscoe-tcpm-inner-space-01 (work in progress), October 2014.

[Ed08] Eddy, W. and A. Langley, "Extending the Space Available for TCP Options", draft-eddy-tcp-loo-04 (work in progress), July 2008.

[Ho11] Honda, M., Nishida, Y., Raiciu, C., Greenhalgh, A., Handley, M., and H. Tokuda, "Is it still possible to extend TCP", Proc. ACM Sigcomm Internet Measurement Conference (IMC), 2011, pp. 181-194.

[Ko04] Kohler, E., "Extended Option Space for TCP", draft-kohler-tcpm-extopt-00 (work in progress), September 2004.

[Ni15] Nishida, Y., "A-PAWS: Alternative Approach for PAWS", draft-nishida-tcpm-apaws-02 (work in progress), Oct. 2015.

[Ra12] Ramaiah, A., "TCP option space extension", draft-ananth-tcpm-tcpoptext-00 (work in progress), March 2012.

[RFC792] Postel, J., "Internet Control Message Protocol", RFC 792, September 1981.

[RFC1812] Baker, F. (Ed.), "Requirements for IP Version 4 Routers," RFC 1812, June 1995.

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

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, September 2001.

[RFC3234] Carpenter, B. and S. Brim, "Middleboxes: Taxonomy and Issues", RFC 3234, February 2002.

[RFC3540] Spring, N., Wetherall, D., and D. Ely, "Robust Explicit Congestion Notification (ECN) Signaling with Nonces", RFC 3540, June 2003.

[RFC5482] Eggert, L., and F. Gont, "TCP User Timeout Option", RFC 5482, March 2009.

[RFC5925] Touch, J., Mankin, A., and R. Bonica, "The TCP Authentication Option", RFC 5925, June 2010.

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

[RFC7323] Borman, D., Braden, B., Jacobson, V., and R. Scheffenegger (Ed.), "TCP Extensions for High Performance", RFC 7323, September 2014.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, December 2014.

[RFC8684] Ford, A., Raiciu, C., Handley, M., Bonaventure, O., and C. Paasch, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 8684, Apr. 2020.

[To22] Touch, J., T. Faber, "TCP SYN Extended Option Space Using an Out-of-Band Segment", draft-touch-tcpm-tcp-syn-ext-opt (work in progress), Oct. 2022.

[Yo11] Yourtchenko, A., "Introducing TCP Long Options by Invalid Checksum", draft-yourtchenko-tcp-loic-00 (work in progress), April 2011.

13. Acknowledgments

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

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

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

Authors' Addresses

Joe Touch
Manhattan Beach, CA 90266 USA

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

Wesley M. Eddy
MTI Systems
US

Email: wes@mti-systems.com

TCP Maintenance and Minor Extensions (TCPM) WG
Internet-Draft
Obsoletes: 675 721 761 813 816 879 896
1078 6013 (if approved)
Updates: 7414 (if approved)
Intended status: Informational
Expires: April 15, 2016

A. Zimmermann
NetApp, Inc.
W. Eddy
MTI Systems
L. Eggert
NetApp, Inc.

October 13, 2015

Moving Outdated TCP Extensions and TCP-related Documents to Historic and Informational Status
draft-ietf-tcpm-undeployed-03

Abstract

This document reclassifies several TCP extensions and TCP-related documents that have either been superseded, have never seen widespread use, or are no longer recommended for use to "Historic" status. The affected RFCs are RFC 675, RFC 721, RFC 761, RFC 813, RFC 816, RFC 879, RFC 896, RFC 1078, and RFC 6013. Additionally, this document reclassifies RFC 700, RFC 794, RFC 814, RFC 817, RFC 872, RFC 889, RFC 964, and RFC 1071 to "Informational" status.

Status of This Memo

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

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

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

This Internet-Draft will expire on April 15, 2016.

1. Introduction

TCP has a long history. Over time, many RFCs have accumulated that describe aspects of the TCP protocol, implementation, and extensions. Some of these have been superseded, are no longer recommended for use, or have simply never seen widespread use.

Section 6 and 7.1 of the TCP roadmap document [RFC7414] already reclassified a number of TCP extensions as "Historic" and describes the reasons for doing so, but it did not instruct the RFC Editor to change the status of these RFCs in the RFC database. The purpose of this document is to do just that.

In addition, this document reclassifies all other documents mentioned in the TCP roadmap that currently have an "Unknown" status to either "Historic" or "Informational".

2. RFC Editor Considerations

The following two sections give a short justification why a specific TCP extension or a TCP-related document is being reclassified as "Historic" or "Informational". In addition, the letter code after an RFC number indicates from which original status a particular RFC is changed to "Historic" or "Informational" (see BCP 9 [RFC2026] for an explanation of these categories):

S - Standards Track (Proposed Standard, Draft Standard, or Internet Standard)

E - Experimental

I - Informational

H - Historic

B - Best Current Practice

U - Unknown (not formally defined)

For the content of the documents itself, the reader is referred either to the corresponding RFC or, for a brief description, to the TCP Roadmap document [RFC7414].

2.1. Moving to "Historic" Status

The RFC Editor is requested to change the status of the following RFCs to "Historic" [RFC2026]:

- [RFC0675] U, "Specification of Internet Transmission Control Program" was replaced by the final TCP specification [RFC0793]
- [RFC0721] U, "Out-of-Band Control Signals in a Host-to-Host Protocol" was a proposal that was not incorporated into the final TCP specification [RFC0793]
- [RFC0761] U, "DoD standard Transmission Control Protocol" was replaced by the final TCP specification [RFC0793]
- [RFC0813] U, "Window and Acknowledgement Strategy in TCP" was incorporated into [RFC1122]
- [RFC0816] U, "Fault Isolation and Recovery" was incorporated into [RFC1122] and [RFC5461]
- [RFC0879] U, "The TCP Maximum Segment Size and Related Topics" was incorporated into [RFC1122] and [RFC6691]
- [RFC0896] U, "Congestion Control in IP/TCP Internetworks" was incorporated into [RFC1122] and [RFC6633]
- [RFC1078] U, "TCP Port Service Multiplexer (TCPMUX)" should be deprecated, because:
  - It modifies the TCP connection establishment semantics by also completing the three-way handshake when a service is not available.
  - It requires all new connections to be received on a single port, which limits the number of connections between two machines.
  - It complicates firewall implementation and management, because all services share the same port number.
  - There are no known client-side deployments.
- [RFC6013] E: "TCP Cookie Transactions (TCPCT)" should be deprecated (although only published in 2011), because:

  * It uses the experimental TCP option codepoints, which prohibits a large-scale deployment.
  * [RFC7413] and [I-D.ietf-tcpm-tcp-edo] are alternatives that have more "rough consensus and running code" behind them.
  * There are no known wide-scale deployments.

2.2. Moving to "Informational" Status

The RFC Editor is requested to change the status of the following RFCs to "Informational" [RFC2026]:

- [RFC0700] U, "A Protocol Experiment", which presents a field report about the deployment of a very early version of TCP

- [RFC0794] U, "Pre-emption", which recommends that operating systems need to manage their limited resources, which may include TCP connection state

- [RFC0814] U, "Name, Addresses, Ports, and Routes", which gives guidance on designing tables and algorithms to keep track of various identifiers within a TCP/IP implementation

- [RFC0817] U, "Modularity and Efficiency in Protocol Implementation", which contains general implementation suggestions

- [RFC0872] U, "TCP-on-a-LAN", which concludes that the fear of using TCP on a local network is unfounded

- [RFC0889] U, "Internet Delay Experiments", which which describes experiments with the TCP retransmission timeout calculation

- [RFC0964] U, "Some Problems with the Specification of the Military Standard Transmission Control Protocol", which points out several specification bugs in the US Military's MIL-STD-1778 document, which was intended as a successor to [RFC0793]

- [RFC1071] U, "Computing the Internet Checksum", which lists a number of implementation techniques for efficiently computing the Internet checksum

3. IANA Considerations

None of the documents moved to "Historic" or "Informational" status have assigned TCP options numbers. Therefore, no IANA actions are required.

4. Security Considerations

This document introduces no new security considerations. Each RFC listed in this document attempts to address the security considerations of the specification it contains.

5. Acknowledgments

The authors thank John Leslie, Pasi Sarolahti, Richard Scheffenegger, Martin Stiemerling, and Joe Touch for their contributions.

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

6. References

6.1. Normative References

[RFC0675] Cerf, V., Dalal, Y., and C. Sunshine, "Specification of Internet Transmission Control Program", RFC 675, DOI 10.17487/RFC0675, December 1974, <http://www.rfc-editor.org/info/rfc675>.

[RFC0700] Mader, E., Plummer, W., and R. Tomlinson, "Protocol experiment", RFC 700, DOI 10.17487/RFC0700, August 1974, <http://www.rfc-editor.org/info/rfc700>.

[RFC0721] Garlick, L., "Out-of-Band Control Signals in a Host-to-Host Protocol", RFC 721, DOI 10.17487/RFC0721, September 1976, <http://www.rfc-editor.org/info/rfc721>.

[RFC0761] Postel, J., "DoD standard Transmission Control Protocol", RFC 761, DOI 10.17487/RFC0761, January 1980, <http://www.rfc-editor.org/info/rfc761>.

[RFC0794] Cerf, V., "Pre-emption", RFC 794, DOI 10.17487/RFC0794, September 1981, <http://www.rfc-editor.org/info/rfc794>.

[RFC0813] Clark, D., "Window and Acknowledgement Strategy in TCP", RFC 813, DOI 10.17487/RFC0813, July 1982, <http://www.rfc-editor.org/info/rfc813>.

[RFC0814] Clark, D., "Name, addresses, ports, and routes", RFC 814, DOI 10.17487/RFC0814, July 1982, <http://www.rfc-editor.org/info/rfc814>.

[RFC0816] Clark, D., "Fault isolation and recovery", RFC 816, DOI 10.17487/RFC0816, July 1982, <http://www.rfc-editor.org/info/rfc816>.

[RFC0817] Clark, D., "Modularity and efficiency in protocol implementation", RFC 817, DOI 10.17487/RFC0817, July 1982, <http://www.rfc-editor.org/info/rfc817>.

[RFC0872] Padlipsky, M., "TCP-on-a-LAN", RFC 872, DOI 10.17487/RFC0872, September 1982, <http://www.rfc-editor.org/info/rfc872>.

[RFC0879] Postel, J., "The TCP Maximum Segment Size and Related Topics", RFC 879, DOI 10.17487/RFC0879, November 1983, <http://www.rfc-editor.org/info/rfc879>.

[RFC0889] Mills, D., "Internet Delay Experiments", RFC 889, DOI 10.17487/RFC0889, December 1983, <http://www.rfc-editor.org/info/rfc889>.

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

[RFC0964] Sidhu, D. and T. Blumer, "Some problems with the specification of the Military Standard Transmission Control Protocol", RFC 964, DOI 10.17487/RFC0964, November 1985, <http://www.rfc-editor.org/info/rfc964>.

[RFC1071] Braden, R., Borman, D., and C. Partridge, "Computing the Internet checksum", RFC 1071, DOI 10.17487/RFC1071, September 1988, <http://www.rfc-editor.org/info/rfc1071>.

[RFC1078] Lottor, M., "TCP port service Multiplexer (TCPMUX)", RFC 1078, DOI 10.17487/RFC1078, November 1988, <http://www.rfc-editor.org/info/rfc1078>.

[RFC6013] Simpson, W., "TCP Cookie Transactions (TCPCT)", RFC 6013, DOI 10.17487/RFC6013, January 2011, <http://www.rfc-editor.org/info/rfc6013>.

6.2. Informative References

[I-D.ietf-tcpm-tcp-edo]
Touch, J. and W. Eddy, "TCP Extended Data Offset Option", draft-ietf-tcpm-tcp-edo-03 (work in progress), April 2015.

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

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <http://www.rfc-editor.org/info/rfc1122>.

[RFC2026] Bradner, S., "The Internet Standards Process -- Revision 3", BCP 9, RFC 2026, DOI 10.17487/RFC2026, October 1996, <http://www.rfc-editor.org/info/rfc2026>.

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

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

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

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <http://www.rfc-editor.org/info/rfc7413>.

[RFC7414] Duke, M., Braden, R., Eddy, W., Blanton, E., and A. Zimmermann, "A Roadmap for Transmission Control Protocol (TCP) Specification Documents", RFC 7414, DOI 10.17487/RFC7414, February 2015, <http://www.rfc-editor.org/info/rfc7414>.

Authors’ Addresses

Alexander Zimmermann
NetApp, Inc.
Sonnenallee 1
Kirchheim 85551
Germany

Phone: +49 89 900594712
Email: alexander.zimmermann@netapp.com

Wesley M. Eddy
MTI Systems
Suite 170, 18013 Cleveland Parkway
Cleveland, OH 44135

Phone: 216-433-6682
Email: wes@mti-systems.com

Lars Eggert
NetApp, Inc.
Sonnenallee 1
Kirchheim 85551
Germany

Phone: +49 89 900594306
Email: lars@netapp.com

TCP Maintenance (TCPM)
Internet-Draft
Intended status: Standards Track
Expires: June 14, 2019

A. Sujeet Nayak
B. Weis
Cisco Systems

December 11, 2018

SHA-2 Algorithm for the TCP Authentication Option (TCP-AO)
draft-nayak-tcp-sha2-03

Abstract

The TCP Authentication Option (TCP-AO) relies on security algorithms to provide connection authentication between the two end-points. This document specifies how to use SHA-256 algorithm and attributes with TCP-AO.

Status of This Memo

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

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

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

This Internet-Draft will expire on June 14, 2019.

Table of Contents

1. Introduction 2
2. Requirements 2
2.1. Requirements Language 2
2.2. Algorithm Requirements 3
3. Algorithms Specified 3
3.1. Key Derivation Functions (KDF) 3
3.1.1. KDF_HMAC_SHA256 4
3.1.2. Tips for User Interfaces Regarding KDFs 4
3.2. MAC Algorithm 5
3.2.1. The Use of HMAC-SHA256-128 5
4. Interaction with TCP 6
5. Security Considerations 6
6. IANA Considerations 7
7. Acknowledgements 7
8. References 7
8.1. Normative References 7
8.2. Informative References 8
Authors' Addresses 8

1. Introduction

[RFC5925] describes TCP-AO mechanism to provide cryptographic authentication and message integrity verification between two end-points of a TCP connection. [RFC5926] specifies HMAC-SHA-1-96 and AES-128-CMAC-96 message authentication codes (MACs) algorithms for TCP-AO.

Although SHA-1 is considered safe for non-digital signature applications at the time of this writing [NIST-SP800-131A], there is a naturally growing demand, especially from the government and service provider community, for protecting their TCP sessions with SHA-2 family of authentication algorithms, which is considered to be relatively stronger. SHA-256, being widely preferred and deployed, provides a reasonable alternative with stronger algorithm and larger MAC length.

This document specifies usage of SHA-256 MAC algorithm on TCP-AO enabled connections. It is a companion to [RFC5925] and [RFC5926].

2. Requirements

2.1. Requirements Language

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

When used in lowercase, these words convey their typical use in common language, and they are not to be interpreted as described in [RFC2119].

2.2. Algorithm Requirements

This document adopts the style and conventions of [RFC5926] in specifying how the use of new data integrity algorithm is indicated in TCP-AO. It indicates a MAC algorithm and a key derivation function (KDF).

The following table indicates the defined SHA-2 algorithm for TCP-AO:

Requirement	Authentication Algorithm
RECOMMENDED	HMAC-SHA-256-128 [RFC2104][FIPS-180-4]

Table 1

Requirement	Key Derivation Function (KDF)
RECOMMENDED	KDF_HMAC-SHA-256

Table 2

3. Algorithms Specified

TCP-AO requires two classes of algorithms to be used on a particular connection namely, Key Derivation Functions (KDF) and Message Authentication Code (MAC) algorithm. Both these classes are generically described in Section 3 in [RFC5926], while focusing specifically on SHA-1 and AES-128 algorithms.

In this document, the same concept is applied to use SHA-256 algorithm.

3.1. Key Derivation Functions (KDF)

KDFs have the following interface:

Traffic_Key = KDF_alg(Master_Key, Context, Output_Length)

where:

Traffic_Key, KDF_alg, Master_Key, Context, Output_Length stand for entities, as described in [RFC5926], Section 3.1.

The KDF algorithm paired with corresponding pseudorandom function (PRF) is:

* KDF_HMAC-SHA-256 based on PRF-HMAC-SHA256 [RFC2104][FIPS-180-4]

It is based on the iteration-mode KDF specified in [NIST-SP800-108]. It uses an underlying PRF with a fixed length output of 256-bits. The KDF generates an arbitrary number of output bits by operating the PRF in a "counter" mode, where each invocation of the PRF uses a different input block, which is differentiated by a block counter.

Each input block is constructed as follows:

(i || Label || Context || Output_Length)

Where:

"||", i, Label, Context, Output_Length stand for entities, as described in [RFC5926], Section 3.1.1.

3.1.1. KDF_HMAC_SHA256

For KDF_HMAC_SHA256:

- PRF for KDF_alg: HMAC-SHA256 [RFC2104][FIPS-180-4]

- Use: HMAC-SHA256(Key, Input)

- Key: Master_Key, configured by user, and passed to KDF

- Input: (i || Label || Context || Output_Length)

- Output_Length: 256 bits

- Result: Traffic_Key, used in MAC function by TCP-AO

3.1.2. Tips for User Interfaces Regarding KDFs

This section provides suggested representations for the KDFs in implementation of user interfaces (UIs). Following these guidelines across common implementations will make interoperability easier and simpler for users deploying TCP-AO.

UIs SHOULD refer to the choice of KDF_HMAC_SHA256 as simply "SHA256".

The IANA registry values reflect this entry.

3.2. MAC Algorithm

Each MAC_alg defined for TCP-AO has three fixed elements as part of its definition:

- KDF_Alg: Name of the TCP-AO KDF algorithm used to generate the Traffic_Key.

- Key_Length: Length, in bits, required for the Traffic_Key used in this MAC.

- MAC_Length: The final length of the bits used in the TCP-AO MAC field. This value may be a truncation of the MAC function's original output length.

As described in [RFC5926], Section 3.2, MACs computed for TCP-AO have the following interface:

MAC = MAC_alg(Traffic_Key, Message)

The MAC_alg for generating MAC, as used by TCP-AO:

* HMAC-SHA256-128 based on [RFC2104] and [FIPS-180-4]

HMAC-SHA256 produces 256 bits output. The MAC output is then truncated to provide a reasonable trade-off between security and message size, for fitting into the TCP-AO option field. As recommended in [RFC2104], Section 5, the HMAC-SHA256 is truncated to 128 bits.

3.2.1. The Use of HMAC-SHA256-128

The three fixed elements for HMAC-SHA256-128 are:

- KDF_Alg: KDF_HMAC_SHA256

- Key_Length: 256 bits

- MAC_Length: 128 bits

For:

MAC = MAC_alg (Traffic_Key, Message)

HMAC-SHA256-128 for TCP-AO has the following values:

- MAC_alg: HMAC-SHA256

- Traffic_Key: Variable; the result of the KDF

- Message: The message to be authenticated, as specified in [RFC5925], Section 5.1

4. Interaction with TCP

As described in [RFC5925], Section 7.6, TCP option space is most critical in SYN segments. As compared to 96-bit Mac length of [RFC5925], using a 128-bit MAC length increases the TCP-AO space from 16 bytes to 20 bytes. Since 9 bytes of space was already available in the SYN segment (9 bytes further reduces to 5 in the presence of MSS option), implementors of this document could use it to provide a stronger authentication algorithm for the TCP connections.

For non-SYN segments, TCP-AO with 128-bit Mac length would use 20 bytes, leaving 20 bytes for other options. Out of these, 10 bytes would be consumed by timestamp, leaving around 10 bytes for a single SACK block. This limit remains the same as described in [RFC5925], Section 7.6.

Another important point to be considered by the implementations is that, in the presence of this feature, since the option space is getting pushed further, care SHOULD be taken to ensure all the options are tightly packed to avoid total options length from spilling beyond the available 40 bytes.

5. Security Considerations

This document inherits all the security considerations of the TCP-AO [RFC5925] and HMAC-SHA-1 related to [RFC5926].

NOTE REGARDING OTHER SHA-2 ALGORITHMS:

In the SHA-2 family, another widely used algorithm in the industry is SHA512. Adopting SHA512 algorithm would mean using a MAC length of 256-bits, as recommended in [RFC2104], Section 5. At the time of writing this document, there is no sufficient space available in the TCP SYN segment to accommodate this large length, without causing backward incompatibility. To avoid this scenario, usage of SHA512 algorithm is deferred, till the time a larger TCP option space evolves.

6. IANA Considerations

As described in [RFC5926], Section 5, IANA has a registry with the following details:

Registry Name: Cryptographic Algorithms for TCP-AO Registration

Procedure: RFC Publication after Expert Review

The following needs to be added to this registry:

Algorithm	Reference
SHA256	This document Number

Table 3

7. Acknowledgements

Bertrand Duvivier, M. Rohit and Srinivas Ramprasad first suggested the need for this work.

8. References

8.1. Normative References

[FIPS-180-4]
FIPS Publication 180-4, "Secured Hash Standard", March 2012.

[NIST-SP800-108]
National Institute of Standards and Technology, "Recommendation for Key Derivation Using Pseudorandom Functions, NIST SP800-108", October 2009.

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

[RFC5925] 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>.

[RFC5926] Lebovitz, G. and E. Rescorla, "Cryptographic Algorithms for the TCP Authentication Option (TCP-AO)", RFC 5926, DOI 10.17487/RFC5926, June 2010, <https://www.rfc-editor.org/info/rfc5926>.

8.2. Informative References

[NIST-SP800-131A]
National Institute of Standards and Technology, "Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths, NIST SP800-131A", January 2011.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

Authors' Addresses

Sujeet Nayak Ammunje
Cisco Systems
Cessna Business Park
Bangalore, Karnataka 560 087
India

Email: sujeetnayak@yahoo.com

Brian Weis
Cisco Systems

Email: bew.stds@gmail.com