TCP Maintenance and Minor Extensions
Internet-Draft
Intended status: Experimental
Expires: 8 January 2021

M. Amend
DT
J. Kang
Huawei

7 July 2020

Multipath TCP Extension for Robust Session Establishment
draft-amend-tcpm-mptcp-robe-00

Abstract

Multipath TCP extends the plain, single-path limited, TCP towards the capability of multipath transmission. This greatly improves the reliability and performance of TCP communication. For backwards compatibility reasons the Multipath TCP was designed to setup successfully an initial path first, after which subsequent paths can be added for multipath transmission. For that reason the Multipath TCP has the same limitations as the plain TCP during connection setup, in case the selected path is not functional.

This document proposes a set of implementations and possible combinations thereof, that provide a more Robust Establishment (RobE) of MPTCP sessions. It includes RobE_TIMER, RobE_SIM, RobE_eSIM and RobE_IPS.

RobE_TIMER is designed to stay close to MPTCP in that standard functionality is used wherever possible. Resiliency against network outages is achieved by modifying the SYN retransmission timer: If one path is defective, another path is used.

RobE_SIM and RobE_eSIM provides the ability to simultaneously use multiple paths for connection setup. They ensure connectivity if at least one functional path out of a bunch of paths is given and offers beside that the opportunity to significantly improve loading times of Internet services.

RobE_IPS provides a heuristic to select properly an initial path for connection establishment with a remote host based on empirical data derived from previous connection information.

In practice, these independent solutions can be complementary used. This document also presents the design and protocol procedure for those combinations in addition to the respective stand-alone solutions.

Status of This Memo

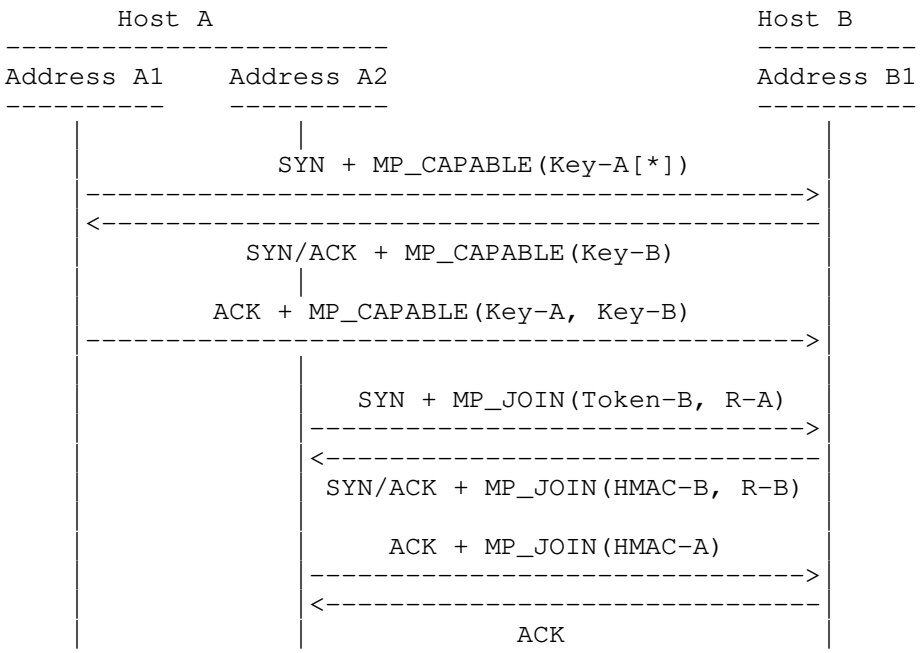
Copyright Notice

Table of Contents

1. Introduction

Multipath TCP Robust Session Establishment (MPTCP RobE) is a set of extensions to regular MPTCP [RFC6824] and its next version [RFC8684], which releases single path limitations during the initial connection setup. Several scenarios require and benefit from a reliable and in time connection setup which is not covered by [RFC6824] and [RFC8684] so far. MPTCP was designed to be compliant with the TCP standard [RFC0793] and introduced therefore the concept of an initial TCP flow while adding subsequent flows after successful multipath negotiation on the initial path. While fulfilling its purpose, MPTCP is however fully dependent on the transmission characteristics of the communication link selected for initiating MPTCP.

Figure 1 shows the traditional way of MPTCP handshaking with a MP_CAPABLE exchanged first, followed when successful negotiated by additional flows engaging MP_JOIN. [RFC6824] and the next MPTCP [RFC8684] differ in that a Key-A is sent with the first MP_CAPABLE or not.



[*] Key-A in the first MP-capable is related to RFC6824 only and does not exist in RFC8684.

Figure 1: MPTCP connection setup

Multipath TCP itself enables hosts to exchange packets belonging to a single connection over several paths. Implemented in mobile phones (UEs), these paths are usually assigned to different network interfaces within the UE and corresponds to different networks such as cellular and WiFi. The path or network interface for initiating the initial subflow setup is most often provided by the operation system of the UE. For example, if a cellular connection and WiFi are present in a mobile phone, WiFi is usually the interface offered to initiate the MPTCP session.

This design falls short in situations where the default path does not provide the best performance compared to other available paths. In a worst case the default path is not even capable of setting up the initial flow letting any other functional path unused. For example, if the WiFi signal is weak, broken or cannot forward traffic to the destination, the establishment of the subflow will be delayed or impossible. This in turn, leads to a longer startup delay or no communication at all for services using MPTCP even if other functional paths are available. Even in scenarios where all paths are functional but services would benefit from a setup over the path with the lowest latency, MPTCP has no mean to support this demand.

It can be concluded, that sequential path establishment relying with an initial path establishment over an external given default route will result in experience reduction when using MPTCP. So this document proposes solutions to overcome the aforementioned limitations and provides a more robust connection setup compared to traditional MPTCP.

RobE_SIM and RobE_eSIM aims to overcome the limitations of [RFC6824] and [RFC8684], using one initial flow and introduces the concept of potential initial flows triggered simultaneously. Potential initial flows gives the freedom to use more than one path to request multipath capability and select the initial flow at a later point. Potential initial flow mechanisms and the gain of robustness and performance over the traditional MPTCP connection setup are evaluated in [RobE_slides] and [RobE_paper]. RobE_SIM is a break-before-make mechanism, guaranteeing at least the robust connection establishment, however the RobE_eSIM reuses every potential initial flow request to combine it with less overhead and accelerated multipath availability, leveraging a new MPTCP option MP_JOIN_CAP. From a standardization perspective, the RobE_SIM is fully compliant with [RFC6824] and [RFC8684] and is herein more of a descriptive and procedural nature. The RobE_eSIM requires a new MPTCP option but with the potential to significantly improve the MPTCP experience.

For the limitation of the default initial path, RobE_IPS makes no changes to standard MPTCP procedure and improves the performance of connection establishment by introducing an initial path selection strategy and required algorithms. The input for strategy and algorithms is the transmission status information which represents the transmission performance of each available path or network interface. The transmission status information is characterized by at least one of the parameters: signal strength, throughput, round-trip time (RTT) and link success rate. In this way, a path with better transmission performance can be learned and determined and the respective network interface can be used for connection establishment.

The most simple approach for a robust MPTCP session establishment is RobE_TIMER, iterating the process of initial path establishment over all available paths, if the previous try has failed. Triggering a new try on a next path is depending on an expiration timer, preferably re-use TCP's in-built expiration timer.

Table 1 summarizes the impact of RobE_TIMER, RobE_SIM, RobE_eSIM and RobE_IPS compared to [RFC6824] and [RFC8684].

Scenario	MPTCP	RobE_TIMER	RobE_SIM	RobE_eSIM	RobE_IPS
IP packet loss	Delayed connection	In the scope of timer	No impact	No impact	Delayed connection
IP broken	No connection	In the scope of timer	No impact	No impact	No connection
IP setup duration dependency	Default route	Default route (+ path 1..n)	Fastest path	Fastest path	Selected path
MP availability + duration	MP_CAPABLE HS + MP_JOIN HS	sum_1..n(MP_CAPABLE_n HS) + MP_JOIN HS	MP_CAPABLE HS + MP_JOIN HS	max(MP_CAPABLE_1 .. MP_CAPABLE_n HS)	MP_CAPABLE HS + MP_JOIN HS
Guaranteeing session setup	Depends on the default route	Yes	Yes	Yes	Depends on selection

Table 1: Overview RobE features during initial connection setup

| IP: Initial Path; MP: Multi-Path; HS: Handshake

1.1. Terminology

This document makes use of a number of terms that are either MPTCP-specific or have defined meaning in the context of MPTCP, as follows:

Path: A sequence of links between a sender and a receiver, defined in this context by a 4-tuple of source and destination address/port pairs.

Subflow: A flow of TCP segments operating over an individual path, which forms part of a larger MPTCP connection. A subflow is started and terminated similar to a regular TCP connection.

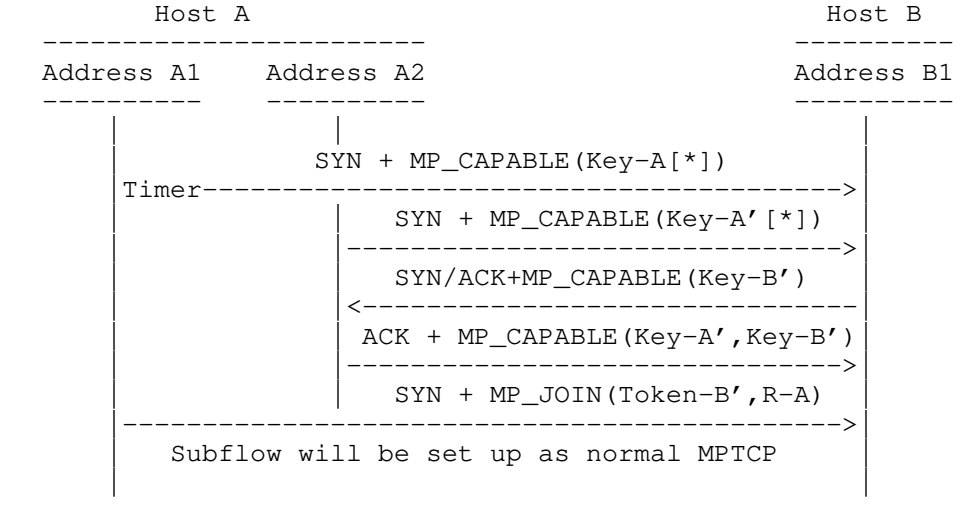
2. Implementation without MPTCP protocol adaptation

RobE_TIMER, RobE_SIM and RobE_IPS are compatible with the current MPTCP protocol definitions in [RFC6824] and [RFC8684] but may be lack of the full optimization potential which require protocol adaptation in Section 3. Following sections will describe them in detail.

2.1. Re-transmission Timer(RobE_TIMER)

In RobE_TIMER, a new connection is initiated by sending a SYN+MP_CAPABLE along the initial path. If this path is functional, the solution will perform identical to classic MPTCP: the initial flow will be established, and subsequent flows can be created afterwards. If however the initial path is faulty, the retransmission will be triggered on another path. This path might circumvent the dysfunctional network, and allow the client to create an initial subflow. The first path is now seen as a subsequent path and the client sends SYN+MP_JOIN messages to create a subsequent flow.

In high latency networks, the initial SYN+MP_CAPABLE might be delayed until the client retries on another path. Once the second SYN arrives at the server, it will try to complete the three-way handshake. If the first SYN was delayed by more than the retransmission time plus half a Round Trip Time (RTT) of the second path, it will arrive at the server after the second SYN. The server could now treat the segment as obsolete and drop it.



[*] Key-A in the first MP-capable is related to RFC6824 only and does not exist in RFC8684.

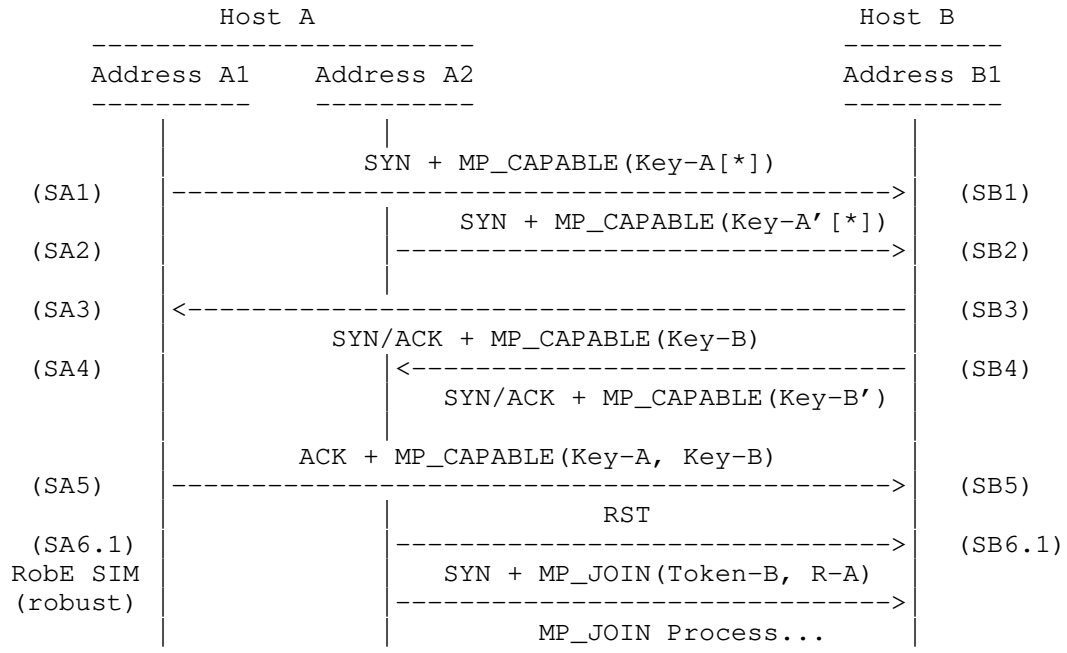
Figure 2: The Robe_TIMER Solution

Immediately after sending the final ACK of the initial handshake, subflows are established on the remaining paths as defined in [RFC6824] and [RFC8684]

[Notes: How to set the Timer is TBD. If there is the case that the first SYN on default path arrives earlier than that from the second path, the MPTCP connection will be initialized on the path of the first SYN. The server could treat the second SYN as obsolete and drop it.]

2.2. Simultaneous Initial Paths Simple Version (RobE_SIM)

RobE_SIM is a sender only implementation and no negotiation is required. In RobE_SIM, the MPTCP connection setup benefits from the fastest path. As shown in Figure 3, host A initiates the connection handshake on more than one path independently (SA1 and SA2). The paths selected for RobE_SIM and referred to as potential initial flows, can belong to the number of interfaces on the device or a subset selected on experience. When Host A receives the first SYN/ ACK back from Host B (SA3), the path carrying this message is identified as the normal initial path. Host A sends then immediately a TCP RST message (SA6.1) on any other path used for simultaneous connection setup causing an immediate termination of assigned flows (break-before-make). The terminated ones are merged as subsequent subflows following the JOIN procedure described in [RFC6824] and [RFC8684]. The process is equivalent to any other scenario where the SYN/ACK arrives on an other path than depicted in Figure 3.



[*] Key-A in the first MP-capable is related to RFC6824 only and does not exist in RFC8684.

Figure 3: MPTCP RobE_SIM Connection Setup

2.3. Heuristic Initial Path Selection (RobE_IPS)

2.3.1. Architecture

Figure 4 provides the architecture for RobE_IPS and employs an "Initial Path Selection" logic which can be integrated into the MPTCP stack or exists as an isolated module in the terminal. The IPS logic has access to a set of transmission status information for each available path or its belonging network interfaces. When an application starts a first communication, IPS selects based on the available path transmission characteristics the path with the highest probability to succeed.

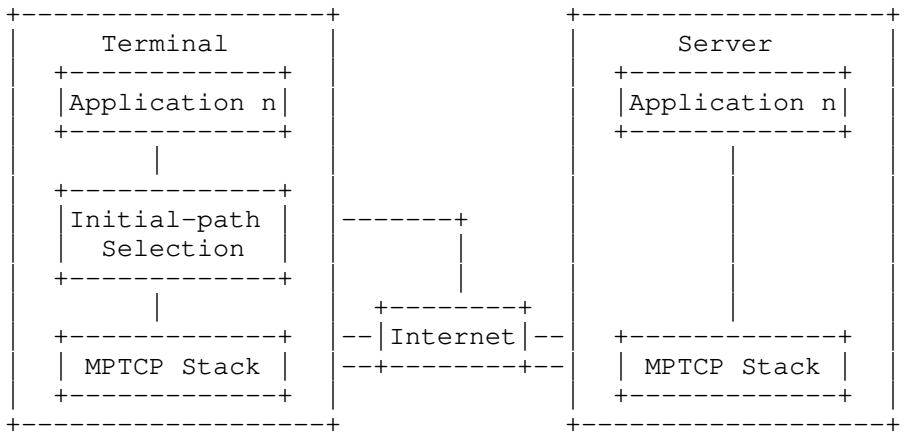


Figure 4: Architecture for Initial-path Selection

2.3.2. Typical Scenarios

Two typical RobE_IPS scenarios are presented in this section. Figure 5 shows that the "Initial Path Selection" logic executed for each MPTCP connection establishment. On the other hand Figure 6 describes that "Initial Path Selection" in case no path information are available. Considering the fact that no heuristics are given before a recent MPTCP connection was established, the default initial path can be adopted. Further combinations and implementations with more or less sophisticated heuristics are possible.

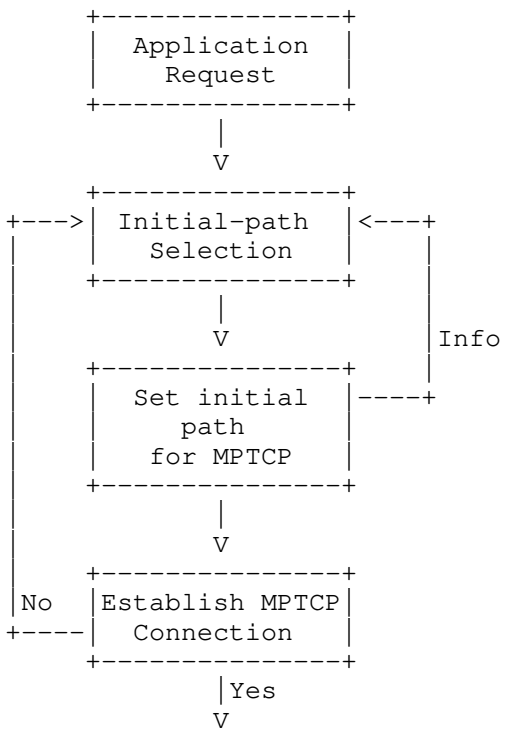


Figure 5: RobE_IPS for each connection establishment

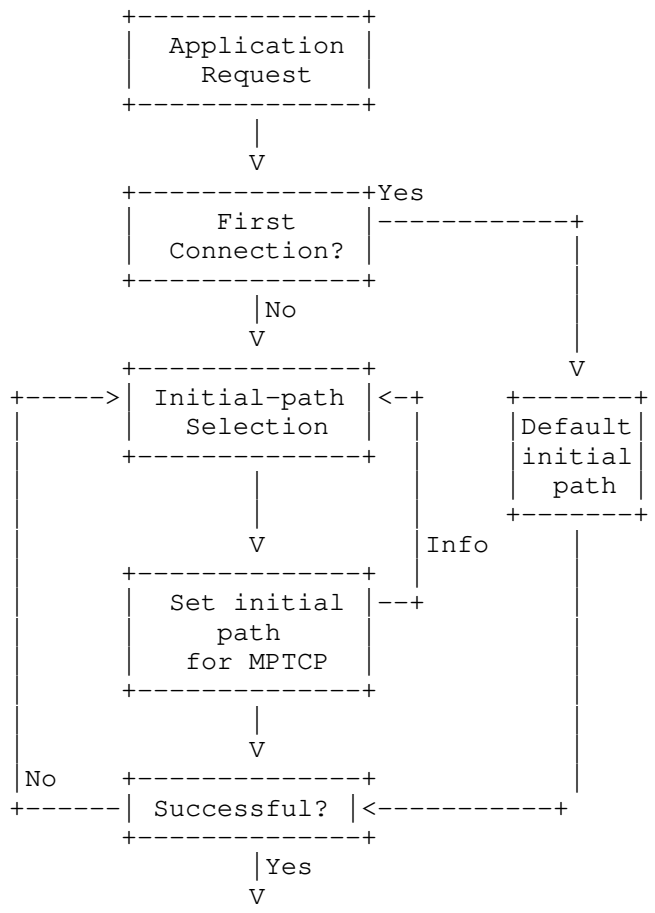


Figure 6: RobE_IPS using default route when no meaningful heuristic available

Figure 7 shows the process flow of "Initial Path Selection". Upon a request from an application, the IPS logic will acquire transmission status information which represents the transmission performance of each available path or network interface and evaluate it. The transmission status information is characterized by at least one of the parameters: signal strength, throughput, round-trip time (RTT) and link success rate. In this way, the path with the best transmission performance can be determined and used for connection establishment.

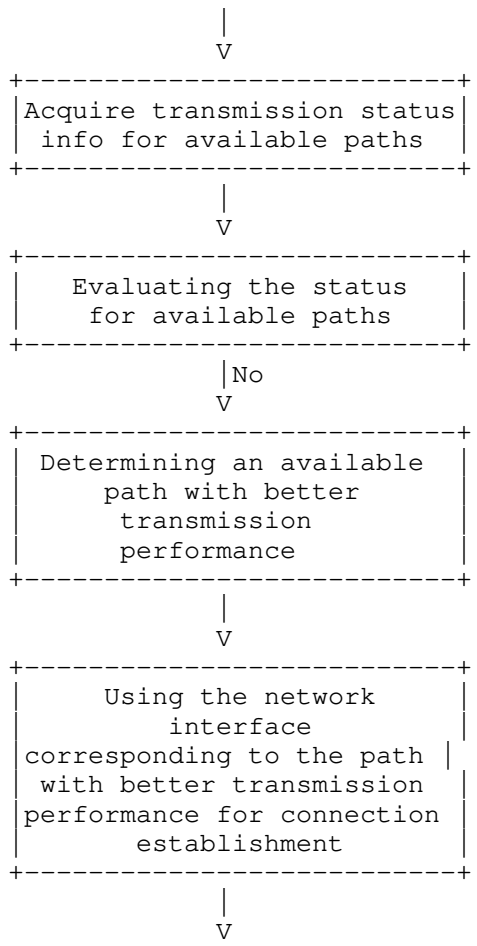


Figure 7: Implementation process for Initial Path Selection

2.3.3. Path decision information

The level of heuristic can be mainly divided into three layer: application level, transport-layer level and link-layer level based on the information acquisition method. For example, RTT can be calculated for each path within a MPTCP connection and belongs thereof to the transport-layer level. The transmission status information for each available path SHOULD be characterized by at least one of the parameters: signal strength, throughput, RTT and link success rate. Application level information are more seen for statistical purposes.

* Application level: application name, domain name, port number and location.

* Transport-layer level: RTT, CWND, Error rate.

2.3.4. Initial Path Selection use local RTT information

Figure 8 presents a "Initial Path Selection" logic based on RTT, e.g. assuming two paths over LTE and WiFi access. RTT calculation on the transport layer usually reflect the time when an information is sent and an related acknowledgment received. For an asymmetric usage (e.g. download only) of a communication it might happen that recent RTT calculation is only available on sender side which is possibly not the side which employs the IPS logic. A solution for this can be found in Section 3.2. Instead of using the most recent RTT value of a path a filtered value consisting of several measured RTTs can be used. A RTT can also be derived from link layer information but may has a limited meaning when it does not picture the end-to-end latency.

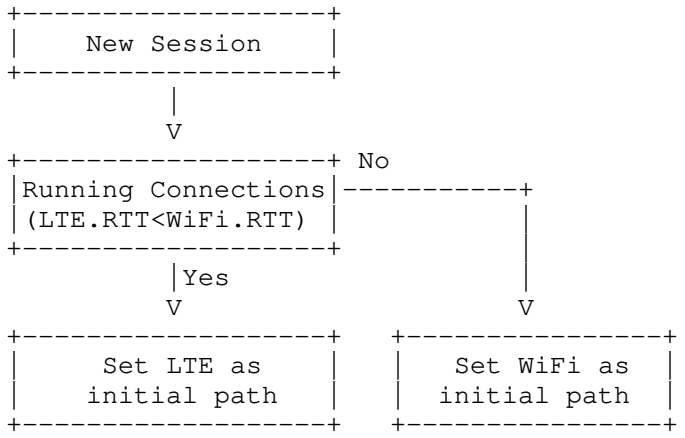


Figure 8: Initial-path Selection based on RTT

2.4. Combination of RobE_SIM and RobE_IPS

In an implementation, a single solution may not be sufficient to get an expected behavior. Combination of approaches to improve robustness is recommended therefore. Figure 9 shows the combination of RobE_SIM and RobE_IPS. RobE_SIM can be used at the very beginning when the sender is absent of path information followed by RobE_IPS for consecutive connections.

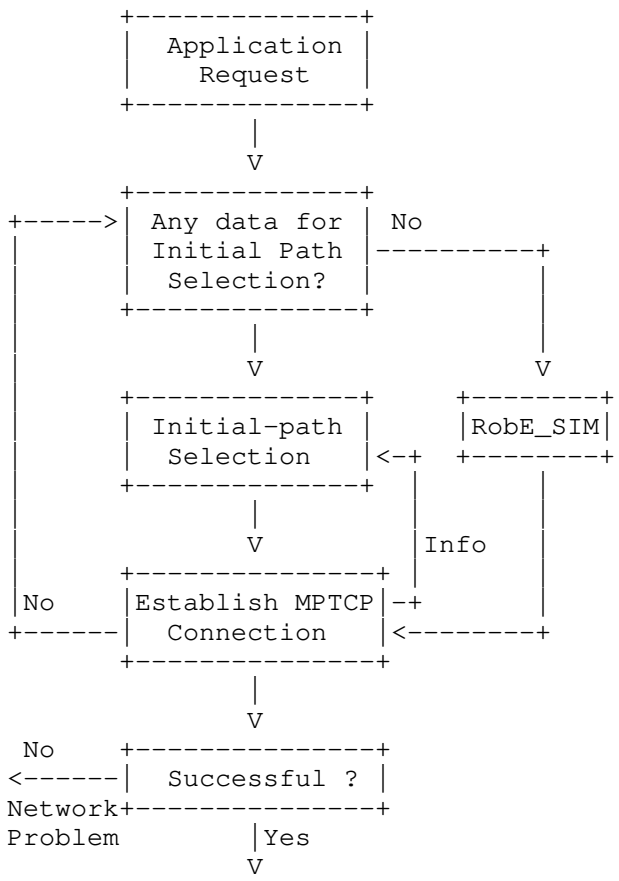


Figure 9: Combination of RobE_SIM and RobE_IPS

2.5. Combination of RobE_TIMER and RobE_IPS

Since RobE_IPS solely does not guarantee that session can be set up on the selection of initial path, it can also be combined with RobE_TIMER which generates less overhead compared to the combination with RobE_SIM in Section 2.4 and guarantees session setup. RobE_TIMER can be introduced to optimize the control of path switching when the initial path selected by RobE_IPS is dysfunctional. When the system enables RobE_IPS and uses the selected initial path for session establishment,it sets the timer for path switching. When timer is expired, the system will change to another path to re-establish connection according to Section 2.1.

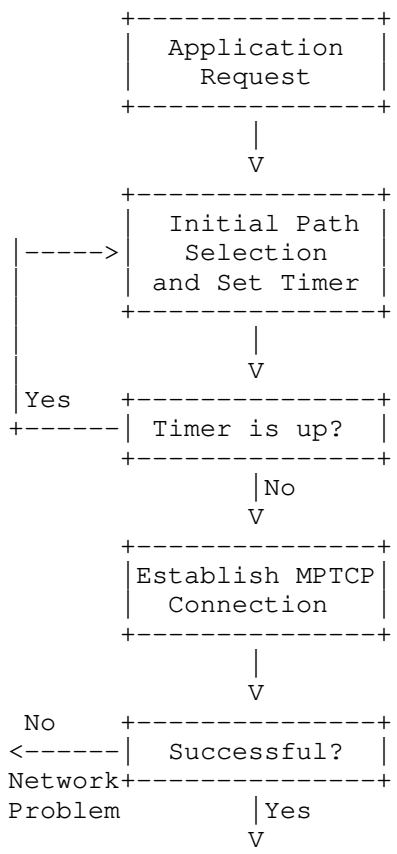


Figure 10: Combination of RobE_Timer and RobE_IPS

3. Implementation with Bi-directional MPTCP Support

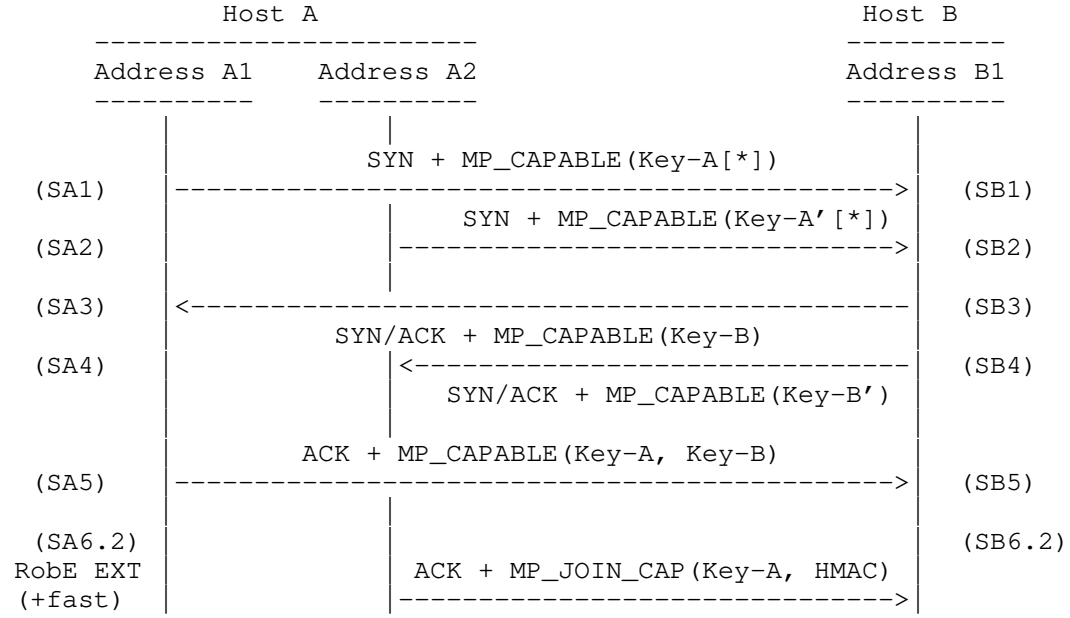
Solutions which requires bi-directional support between two MPTCP hosts promise to have better and possibly more features. However, they cannot be defined without extending current standards in [RFC6824] and [RFC8684]. The RobE_SIM and RobE_IPS approach are both capable of profit from an explicit support of the remote end host and defined within this section.

3.1. Simultaneous Initial Paths Extended Version (RobE_eSIM)

RobE_eSIM extends RobE_SIM by reusing the potential initial flows. This eliminates the overhead from RobE_SIM by introducing a new option MP_JOIN_CAP and accelerate the transmission speed by early availability of multiple paths. Further it relaxes the dependency on a reliable third ACK of the 3-way handshake in [RFC8684]. Remote endpoint support can be negotiated in two ways, an implicit in Section 3.1.1 or explicit in Section 3.1.2.

3.1.1. RobE_eSIM implicit Negotiation and Procedure

Similar to RobE_SIM in Section 2.2, the establishment process of [RFC6824] or [RFC8684] is applied independently on multiple path simultaneously. In Figure 11 this is shown in SA1 and SA2. The first path which returns a SYN/ACK (e.g. SA3) is selected as the initial path and proceed with the traditional establishment process (SA5). Any other path which has to send the final ACK of the 3-way handshake includes a new option MP_JOIN_CAP (see definition in Section 3.1.3.2) instead of a MP_CAPABLE (SA6.2).



[*] Key-A in the first MP-capable is related to RFC6824 only and does not exist in RFC8684.

Figure 11: MPTCP RobE_eSIM implicit Connection Setup

Following the possible process in Figure 11, two further constellations are imaginable and elaborated below.

1. In the flow diagram Figure 11, A1<->B1 is assumed to be the initial flow. A2<->B1 shall be recycled and the ACK is sent with MP_JOIN_CAP. Furthermore, the MP_CAPABLE arrives first at Host B (SB5) and the MP_JOIN_CAP afterwards (SB6.2). When the MP_JOIN_CAP is received, Host B has to iterate over the connection list once (like MP_JOIN) and check for Key-A availability. If a Key-A connection is found, this one is validated against the HMAC value. The validation has two reasons: first, several Key-A can exist, because different hosts may choose the same Key-A by accident. Furthermore, no one can join a connection by just recording/brute-forcing Key-A and duplicating the request.

2. Like above, but MP_JOIN_CAP arrives before last MP_CAPABLE at Host B

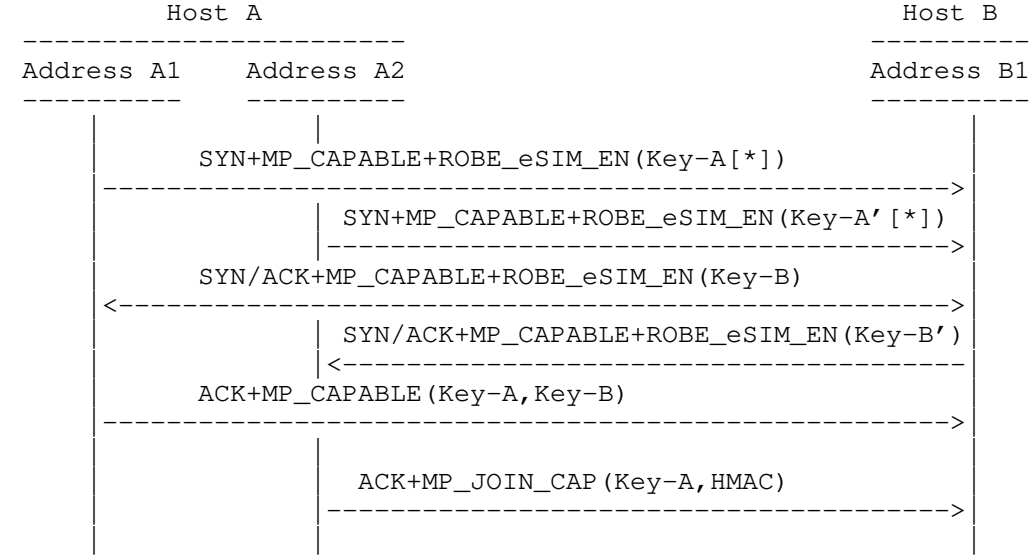
   * [RFC8684]; Based on Key-A, Host B will iterate over the connection list, but it will not find a match, because Key-A of the previous selected initial flow (SA3, SA5) has not arrived yet. So it will continue with a fast iteration only over the connections which are still in establishment phase using the 10 bit Key-B fast hash (crc16(Key-B) & 0x3FF). If it matches against a (precomputed) existing Key-B_fast_hash in the connection list, it will validate the request using the HMAC(Key-A+B+B') to ensure legitimation. If successful, both, the initial flow and the MP_JOIN_CAP flow, can be immediately established. This is true, because without the knowledge of Key-B, Host A could not calculate the HMAC. So it is clear, that Host A had received the SYN/ACK (SB3). This also mitigates the exchange of a reliable ACK during the handshake process. MPTCP sends the Key-A only with the last ACK and therefore prevents subsequent flow establishment until successful reception at Host B. Using RobE_EXT, the reception of a MP_JOIN_CAP ([RFC8684]) is sufficient to establish both, the path carrying Key-B and Key-B'.

   * [RFC6824]; Can match based on Key-A, same effort as for a MP JOIN.

3. A2<->B1 is selected as initial flow, because the respective SYN/ ACK returns earlier at Host A. It is the same as above, just the other way round.

3.1.2. RobE_eSIM explicit Negotiation and Procedure

The process of an explicit negotiation of RobE_eSIM follows Figure 11 but uses the ROBE_eSIM_EN option Figure 13 additionally during the handshake procedure.



[*] Key-A in the first MP-capable is related to RFC6824 only and does not exist in RFC8684.

Figure 12: MPTCP RobE_eSIM explicit Connection Setup

3.1.3. Protocol Adaptation

3.1.3.1. ROBE_eSIM_EN Option

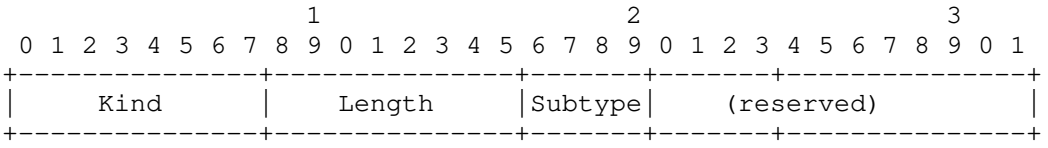


Figure 13: ROBE_eSIM_EN_OPTION

3.1.3.2. MP_JOIN_CAP Option

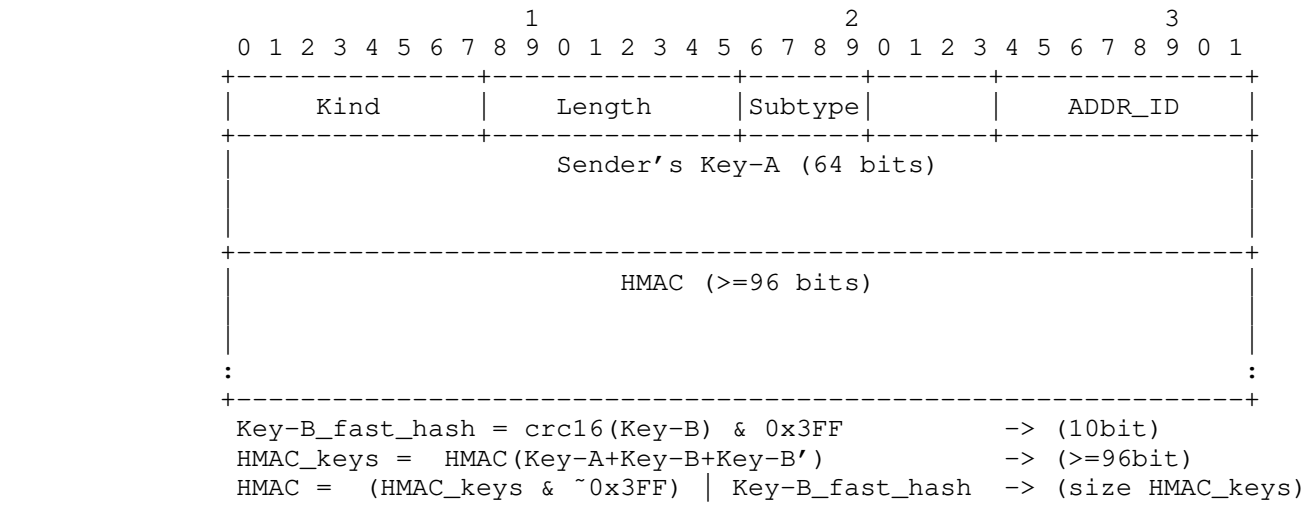


Figure 14: MP_JOIN_CAP

Computational effort on receiver side is most often expected to be the same as with MP_JOIN. Key-A ensures identification of related flows Key-B_fast_hash enables MP session even when selected initial flow is not fully established yet (slight computational overhead). HMAC authenticates relationship of initial and potential initial flows.

3.1.4. Fallback Mechanisms

3.1.4.1. Fallback mechanism for implicit RobE_eSIM

[TBD]

3.1.4.2. Fallback mechanism for explicit RobE_eSIM

This mechanism considers that both sides support MPTCP capability but the receiver does not equipped with RobE_eSIM. MPTCP session with RobE_eSIM negotiation will seamlessly fallback to normal MPTCP process.

[Requires further check how an unaware Host B reacts on possible ROBE_eSIM_EN; Ignore or RST? See also RFC6824 Sec. 3.6 "Should fallback [...] the path does not support the MPTCP options"]

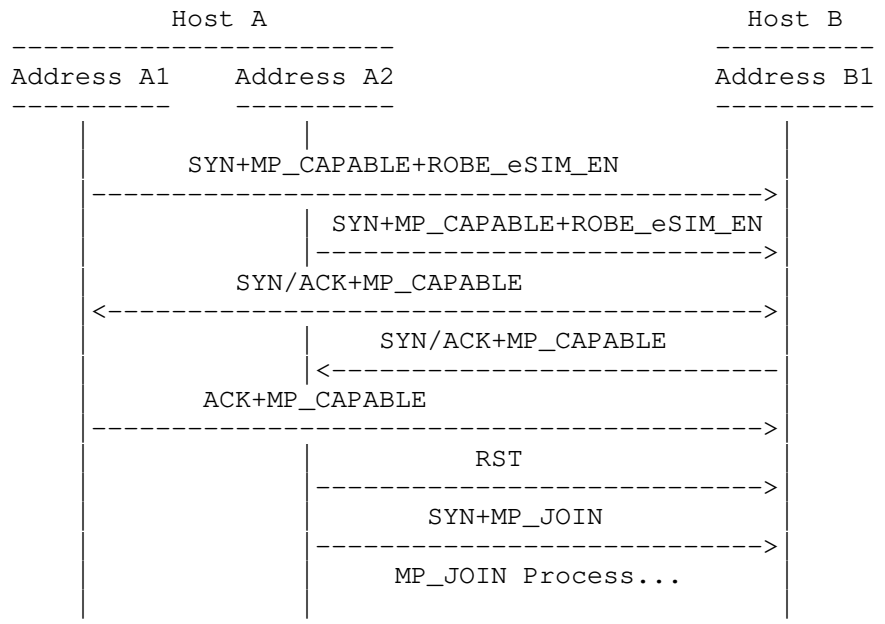


Figure 15: Fallback to MPTCP when missing RobE_eSIM support

3.1.4.3. Fallback to regular TCP when missing MPTCP support

When the receiver is not MPTCP enabled, MPTCP session with RobE_eSIM negotiation will seamlessly fallback to regular process which is illustrated in this section.

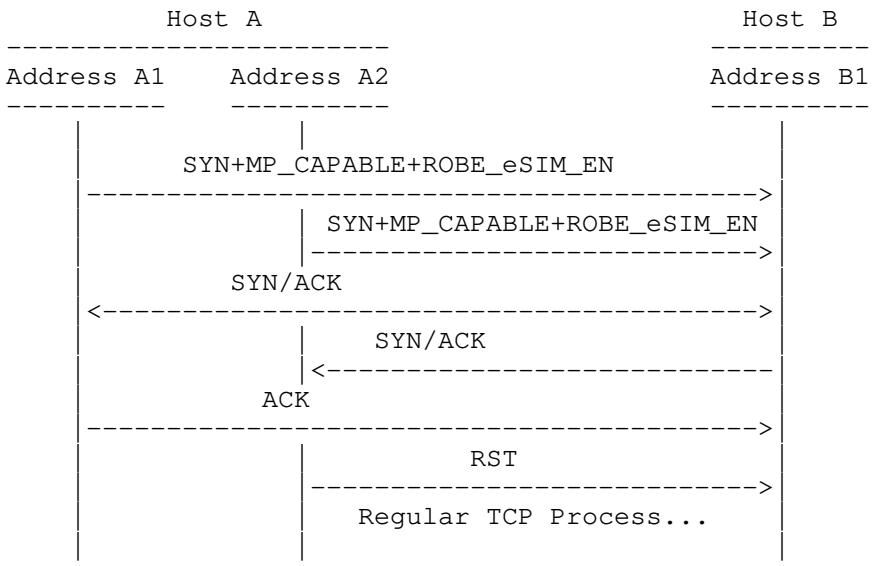
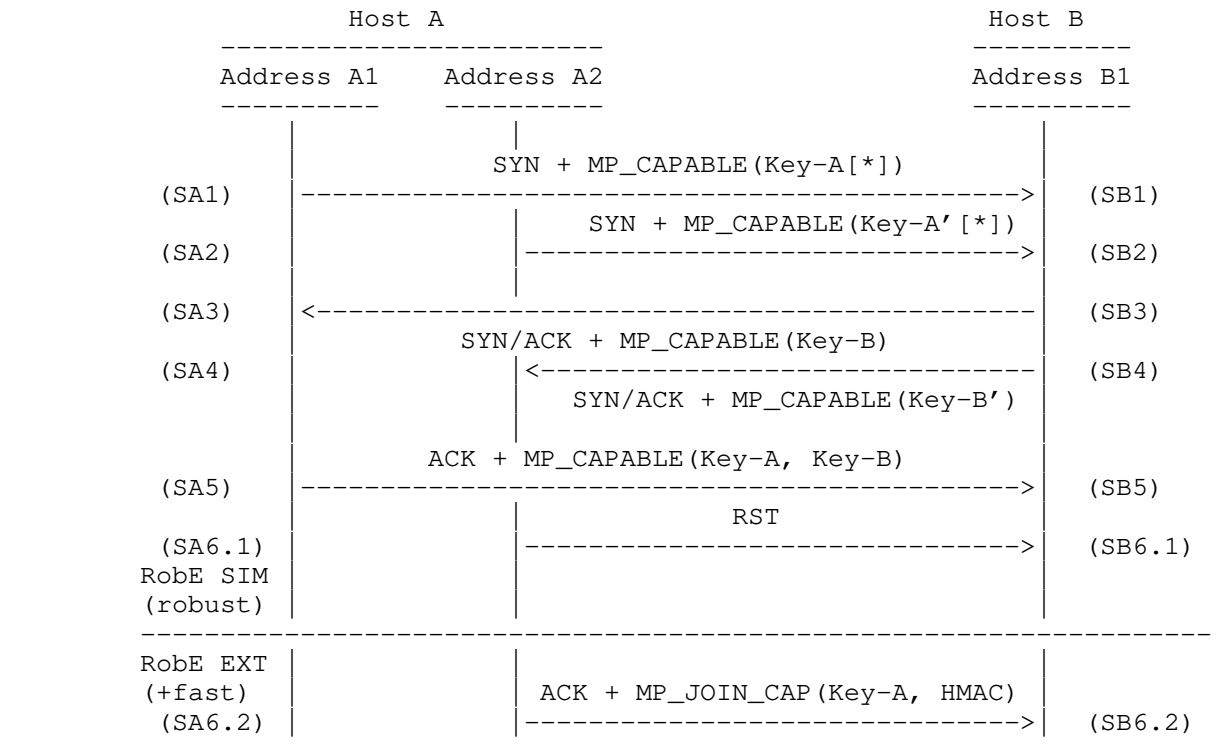


Figure 16: Fallback to TCP without MPTCP support

3.1.5. Comparison Robe_SIM and RobE_eSIM

Potential initial flows in RobE_SIM Section 2.2 and RobE_eSIM Section 3.1 guarantee MPTCP session establishment if at least one selected path for session establishment is functional. Figure 17 makes the differences between both approaches visible and points to the latest decision possibility during session setup when RobE_SIM or RobE_eSIM can be selected. Until SA5 in Figure 17 traditional MPTCP connection setup is independently applied on multiple paths simultaneously and offers to select the initial flow later (potential initial flows). The final decision which path is selected as the main one and the handling of the remaining flow(s) differs in SA6.1 when RobE_SIM is applied or instead SA6.2 RobE_eSIM.



[*] Key-A in the first MP-capable is related to RFC6824 only and does not exist in RFC8684.

Figure 17: MPTCP RobE_SIM and RobE_eSIM connection setup

3.1.6. Security Consideration

[Tbd, however no differences to [RFC6824] and {{RFC8684} are expected]

3.2. Heuristic Initial Path Selection with remote RTT Measurement

3.2.1. Description

Usually the path RTT can be determined by a time difference between sending a package and an ACK and is integrated into the TCP protocol. For asymmetric transmission, the latest RTT for TCP flows is calculated by the side which sends data at latest and possible does not correspond to the site which employs RobE_IPS. This problem is already elaborated in Section 2.3.4 and can be solved by transmitting the RTT information per subflow. The negotiation procedure is depicted in Figure 18 and uses the MPTCP option L_RTT_EN defined in Section 3.2.2.

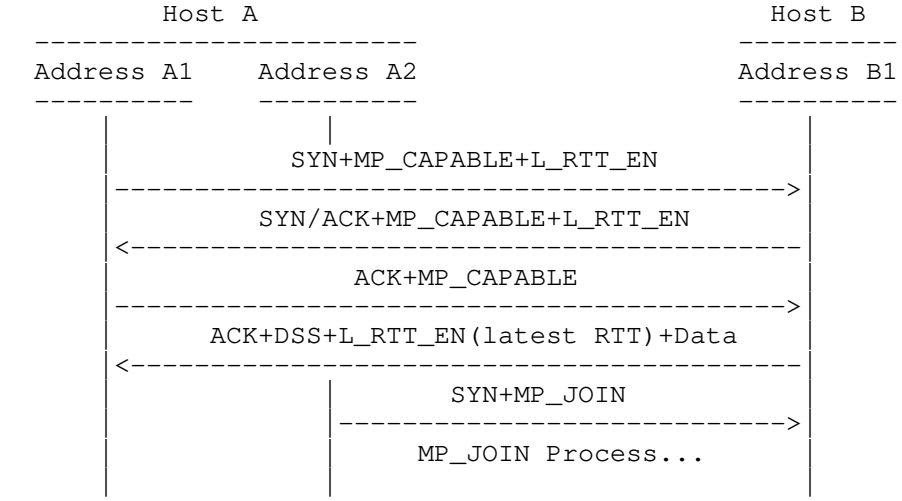


Figure 18: Negotiation procedure for RTT exchange

A successful negotiation allows the exchange of the measured RTT value from one subflow of a MPTCP host to another using the "Latest RTT" field within the L_RTT_EN option.

3.2.2. Protocol Adaptation

Calculating the "Latest RTT" by a remote host in an asymmetry transmission scenario should be transferred from remote host to the client running RobE_IPS. So a new MPTCP subtype option named L_RTT_EN is allocated for this function. During the three-way handshake L_RTT_EN is used for negotiation of remote RTT measurement capability between client and server (in Section 3.2.1). When both parts support the usage of remote RTT measurement, the "Latest RTT" field in L_RTT_EN is applied for carrying the value of latest RTT computed by the remote host.

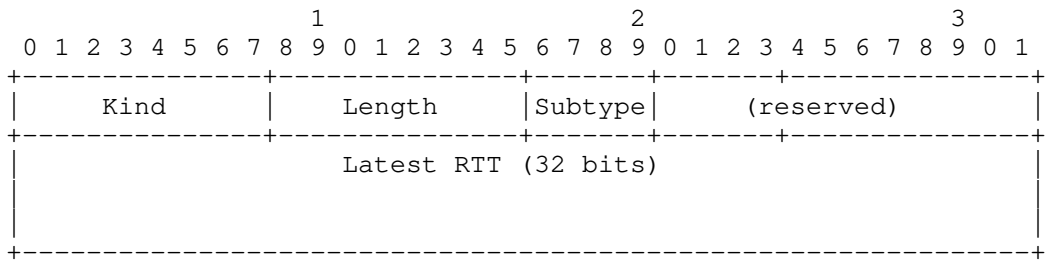


Figure 19: ROBE_L_RTT_EN OPTION

3.2.3. Fallback Mechanism

When the receiver is not L_RTT_EN capable, MPTCP session with L_RTT_EN negotiation will seamlessly fallback to normal MPTCP process.

[TBD, Need same checks as Section 3.1.4.2]

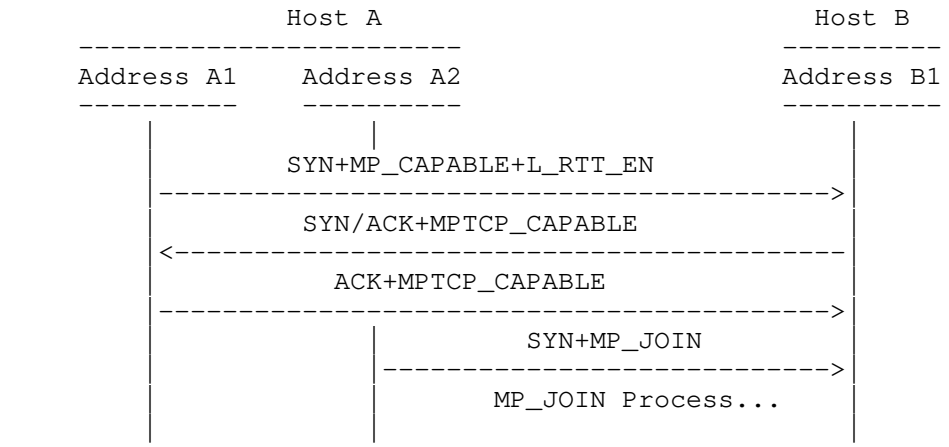


Figure 20: Fallback to MPTCP without RobE_IPS

3.2.4. Security Consideration

[Tbd]

4. IANA Considerations

This document defines three new values to MPTCP Option Subtype as following.

Value	Symbol	Name	Reference
TBD	ROBE_eSIM_EN	RobE_eSIM enabled	Section 3.1
TBD	MP_JOIN_CAP	Join connection directly in RobE_eSIM	Section 3.1
TBD	L_RTT_EN	Server RTT enabled	Section 3.2

Table 2: RobE Option Subtypes

5. References

5.1. Normative References

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

[RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <https://www.rfc-editor.org/info/rfc6824>.

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

5.2. Informative References

[RobE_paper] Amend, M., Rakocevic, V., Matz, A.P., and E. Bogenfeld, "RobE: Robust Connection Establishment for Multipath TCP", ANRW '18 p. 76-82, 16 July 2018, <http://doi.acm.org/10.1145/3232755.3232762>.

[RobE_slides] Amend, M., Matz, A.P., and E. Bogenfeld, "A proposal for MPTCP Robust session Establishment (MPTCP RobE)", IETF 99 Multipath TCP WG session, 18 July 2017, <https://datatracker.ietf.org/meeting/99/materials/slides-99-mptcp-a-proposal-for-mptcp-robust-session-establishment-mptcp-robe-01>.

Authors' Addresses

Markus Amend
Deutsche Telekom
Deutsche-Telekom-Alle 9
64295 Darmstadt
Germany

Email: Markus.Amend@telekom.de

Jiao Kang
Huawei
D2-03,Huawei Industrial Base
Shenzhen
Guangdong, 518129
China

Email: kangjiao@huawei.com

TCPM Working Group
Internet-Draft
Intended status: Experimental
Expires: May 4, 2021

C. Gomez
UPC
J. Crowcroft
University of Cambridge

October 31, 2020

TCP ACK Rate Request Option
draft-gomez-tcpm-ack-rate-request-01

Abstract

TCP Delayed Acknowledgments (ACKs) is a widely deployed mechanism that allows reducing protocol overhead in many scenarios. However, Delayed ACKs may also contribute to suboptimal performance. When a relatively large congestion window (cwnd) can be used, less frequent ACKs may be desirable. On the other hand, in relatively small cwnd scenarios, eliciting an immediate ACK may avoid unnecessary delays that may be incurred by the Delayed ACKs mechanism. This document specifies the TCP ACK Rate Request (TARR) option. This option allows a sender to indicate the ACK rate to be used by a receiver, and it also allows to request immediate ACKs from a receiver.

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 May 4, 2021.

Table of Contents

1. Introduction

Delayed Acknowledgments (ACKs) were specified for TCP with the aim to reduce protocol overhead [RFC1122]. With Delayed ACKs, a TCP delays sending an ACK by up to 500 ms (often 200 ms, with lower values in recent implementations such as ~50 ms also reported), and typically sends an ACK for at least every second segment received in a stream of full-sized segments. This allows combining several segments into a single one (e.g. the application layer response to an application layer data message, and the corresponding ACK), and also saves up to one of every two ACKs, under many traffic patterns (e.g. bulk transfers). The "SHOULD" requirement level for implementing Delayed ACKs in RFC 1122, along with its expected benefits, has led to a widespread deployment of this mechanism.

However, there exist scenarios where Delayed ACKs contribute to suboptimal performance. We next roughly classify such scenarios into two main categories, in terms of the congestion window (cwnd) size and the Maximum Segment Size (MSS) that would be used therein: i) "large" cwnd scenarios (i.e. cwnd >> MSS), and ii) "small" cwnd scenarios (e.g. cwnd up to ~MSS).

In "large" cwnd scenarios, increasing the number of data segments after which a receiver transmits an ACK beyond the typical one (i.e. 2 when Delayed ACKs are used) may provide significant benefits. One

example is mitigating performance limitations due to asymmetric path capacity (e.g. when the reverse path is significantly limited in comparison to the forward path) [RFC3449]. Another advantage is reducing the computational cost both at the sender and the receiver, and reducing network packet load, due to the lower number of ACKs involved.

In many "small" cwnd scenarios, a sender may want to request the receiver to acknowledge a data segment immediately (i.e. without the additional delay incurred by the Delayed ACKs mechanism). In high bit rate environments (e.g. data centers), a flow's fare share of the available Bandwidth Delay Product (BDP) may be in the order of one MSS, or even less. For an accordingly set cwnd value (e.g. cwnd up to MSS), Delayed ACKs would incur a delay that is several orders of magnitude greater than the RTT, severely degrading performance. Note that the Nagle algorithm may produce the same effect for some traffic patterns in the same type of environments [RFC8490]. In addition, when transactional data exchanges are performed over TCP, or when the cwnd size has been reduced, eliciting an immediate ACK from the receiver may avoid idle times and allow timely continuation of data transmission and/or cwnd growth, contributing to maintaining low latency.

Further "small" cwnd scenarios can be found in Internet of Things (IoT) environments. Many IoT devices exhibit significant memory constraints, such as only enough RAM for a send buffer size of 1 MSS. In that case, if the data segment does not elicit an application-layer response, the Delayed ACKs mechanism unnecessarily contributes a delay equal to the Delayed ACK timer to ACK transmission. The sender cannot transmit a new data segment until the ACK corresponding to the previous data segment is received and processed.

With the aim to provide a tool for performance improvement in both "large" and "small" cwnd scenarios, this document specifies the TCP ACK Rate request (TARR) option. This option allows a sender to indicate the ACK rate to be used by a receiver, and it also allows to request immediate ACKs from a receiver.

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 [RFC2119].

3. TCP ACK Rate Request Functionality

A TCP endpoint announces that it supports the TARR option by including the TARR option format in packets that have the SYN bit set. In such packets, the values carried by the TARR option format other than Kind, Length and ExID MUST be ignored by the receiving TCP.

TBD1: perhaps two formats (with one codepoint each), a shorter one just to advertise that TARR is supported, and a larger one which is the current TARR option format defined in section 4?

The next two subsections define the sender and receiver behaviors for devices that support the TARR option, respectively.

3.1. Sender behavior

A TCP sender MUST NOT include the TARR option in TCP data segments to be sent if the TCP receiver does not support the TARR option.

A TCP sender MAY request a TARR-option-capable receiver to modify the ACK rate of the latter to one ACK every R full-sized data segments received from the sender. This request is performed by the sender by including the TARR option in the TCP header of a data segment. The TARR option carries the R value requested by the sender (see section 4). For the described purpose, the value of R MUST NOT be zero. The TARR option also carries the N field, which MUST be ignored when R is not set to zero.

When a TCP sender needs a data segment to be acknowledged immediately by a TARR-option-capable receiving TCP, the sender includes the TARR option in the TCP header of the data segment, with a value of R equal to zero. When R is set to zero, the N field of the same option indicates the number of subsequent data segments for which the sender also requests immediate ACKs.

A TCP sender MAY indicate that it has a reordering tolerance of R packets by setting the Ignore Order field of the TARR option to True (see Section 4).

3.2. Receiver behavior

A receiving TCP conforming to this specification MUST process a TARR option present in a received data segment.

When the TARR option of a received segment carries an R value different from zero, a TARR-option-capable receiving TCP MUST modify its ACK rate to one ACK every R full-sized received data segments

from the sender, as long as packet reordering does not occur. When R is different from zero, the receiving TCP MUST ignore the N field of the TARR option.

A TARR-option-capable TCP that receives a TARR option with the Ignore Order field set to True (see Section 4), MUST NOT send an ACK after each reordered data segment. Instead, it MUST continue to send one ACK every R received data segments. Otherwise (i.e., Ignore Order = False), such a receiver will need to send an ACK after each reordered data segment received.

If a TARR-option-capable TCP receives a segment carrying the TARR option with R=0, the receiving TCP MUST send an ACK immediately, and it MUST also send an ACK immediately after each one of the N next consecutive segments to be received. N refers to the corresponding field in the TARR option of the received segment (see Section 4).

4. Option Format

The TARR option has the format and content shown in Fig. 1.

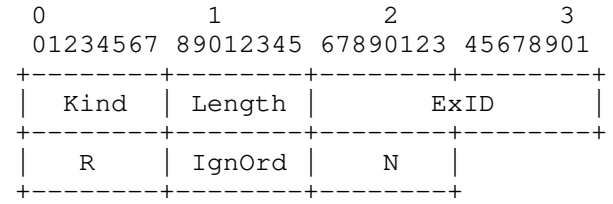


Figure 1: TCP ACK Rate Request option format.

Kind: The Kind field value is TBD.

Length: The Length field value is 7 bytes.

ExID: The experiment ID field size is 2 bytes, and its value is 0x00AC.

R: The size of this field is 1 byte. If all bits of this field are set to 0, the field indicates a request by the sender for the receiver to trigger one or more ACKs immediately. Otherwise, the field carries the binary encoding of the number of full-sized segments received after which the receiver is requested by the sender to send an ACK.

Ignore Order: The size of this field is 1 byte. This field MUST have the value 0x01 ("True") or 0x00 ("False"). When this field is set to True, the receiver MUST NOT send an ACK after each reordered data segment. Instead, it MUST continue to send one ACK every R received data segments.

TBD2: perhaps 7 bits for R and 1 bit for Ignore Order?

N: The size of this field is 1 byte. When R=0, the N field indicates the number of subsequent consecutive data segments to be sent for which immediate ACKs are requested by the sender.

5. IANA Considerations

This document specifies a new TCP option (TCP ACK Rate Request) that uses the shared experimental options format [RFC6994], with ExID in network-standard byte order.

The authors plan to request the allocation of ExID value 0x00AC for the TCP option specified in this document.

6. Security Considerations

TBD

7. Acknowledgments

Bob Briscoe, Jonathan Morton, Richard Scheffenegger, Neal Cardwell, Michael Tuexen, Yuchung Cheng, Matt Mathis, Jana Iyengar, Gorry Fairhurst, and Stuart Cheshire provided useful comments and input for this document. Jana Iyengar suggested including a field to allow a sender communicate its tolerance to reordering. Gorry Fairhurst suggested adding a mechanism to request a number of consecutive immediate ACKs.

Carles Gomez has been funded in part by the Spanish Government through project PID2019-106808RA-I00, and by Secretaria d'Universitats i Recerca del Departament d'Empresa i Coneixement de la Generalitat de Catalunya 2017 through grant SGR 376.

8. References

8.1. Normative References

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <https://www.rfc-editor.org/info/rfc1122>.

[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>.

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

8.2. Informative References

[RFC3449] Balakrishnan, H., Padmanabhan, V., Fairhurst, G., and M. Sooriyabandara, "TCP Performance Implications of Network Path Asymmetry", BCP 69, RFC 3449, DOI 10.17487/RFC3449, December 2002, <https://www.rfc-editor.org/info/rfc3449>.

[RFC8490] Bellis, R., Cheshire, S., Dickinson, J., Dickinson, S., Lemon, T., and T. Pusateri, "DNS Stateful Operations", RFC 8490, DOI 10.17487/RFC8490, March 2019, <https://www.rfc-editor.org/info/rfc8490>.

Authors' Addresses

Carles Gomez
UPC
C/Esteve Terradas, 7
Castelldefels 08860
Spain

Email: carlesgo@entel.upc.edu

Jon Crowcroft
University of Cambridge
JJ Thomson Avenue
Cambridge, CB3 0FD
United Kingdom

Email: jon.crowcroft@cl.cam.ac.uk

NETCONF Working Group
Internet-Draft
Intended status: Standards Track
Expires: 21 February 2021

K. Watsen
Watsen Networks
M. Scharf
Hochschule Esslingen

20 August 2020

YANG Groupings for TCP Clients and TCP Servers
draft-ietf-netconf-tcp-client-server-08

Abstract

This document defines three YANG 1.1 [RFC7950] modules to support the configuration of TCP clients and TCP servers, either as standalone or in conjunction with a stack protocol layer specific configurations.

Editorial Note (To be removed by RFC Editor)

This draft contains placeholder values that need to be replaced with finalized values at the time of publication. This note summarizes all of the substitutions that are needed. No other RFC Editor instructions are specified elsewhere in this document.

Artwork in this document contains shorthand references to drafts in progress. Please apply the following replacements:

* "DDDD" --> the assigned RFC value for this draft

Artwork in this document contains placeholder values for the date of publication of this draft. Please apply the following replacement:

* "2020-08-20" --> the publication date of this draft

The following Appendix section is to be removed prior to publication:

* Appendix A. Change Log

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/.

This Internet-Draft will expire on 21 February 2021.

Table of Contents

1. Introduction

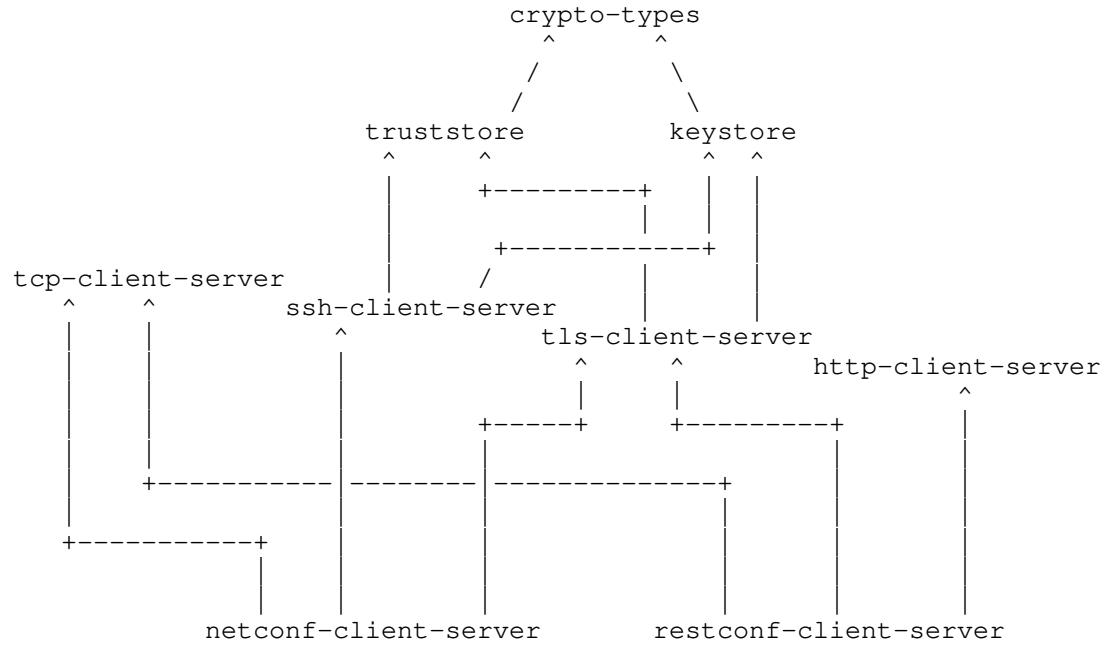
This document defines three YANG 1.1 [RFC7950] modules to support the configuration of TCP clients and TCP servers, either as standalone or in conjunction with a stack protocol layer specific configurations.

1.1. Relation to other RFCs

This document presents one or more YANG modules [RFC7950] that are part of a collection of RFCs that work together to define configuration modules for clients and servers of both the NETCONF [RFC6241] and RESTCONF [RFC8040] protocols.

The modules have been defined in a modular fashion to enable their use by other efforts, some of which are known to be in progress at the time of this writing, with many more expected to be defined in time.

The normative dependency relationship between the various RFCs in the collection is presented in the below diagram. The labels in the diagram represent the primary purpose provided by each RFC. Hyperlinks to each RFC are provided below the diagram.



Label in Diagram	Originating RFC
crypto-types	[I-D.ietf-netconf-crypto-types]
truststore	[I-D.ietf-netconf-trust-anchors]
keystore	[I-D.ietf-netconf-keystore]
tcp-client-server	[I-D.ietf-netconf-tcp-client-server]
ssh-client-server	[I-D.ietf-netconf-ssh-client-server]
tls-client-server	[I-D.ietf-netconf-tls-client-server]
http-client-server	[I-D.ietf-netconf-http-client-server]
netconf-client-server	[I-D.ietf-netconf-netconf-client-server]
restconf-client-server	[I-D.ietf-netconf-restconf-client-server]

Table 1: Label to RFC Mapping

1.2. Specification Language

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

1.3. Adherence to the NMDA

This document in compliant with the Network Management Datastore Architecture (NMDA) [RFC8342]. It does not define any protocol accessible nodes that are "config false".

2. The "ietf-tcp-common" Module

This section defines a YANG 1.1 [RFC7950] module called "ietf-tcp-common". A high-level overview of the module is provided in Section 2.1. Examples illustatrating the module's use are provided in Examples (Section 2.2). The YANG module itself is defined in Section 2.3.

2.1. Data Model Overview

This section provides an overview of the "ietf-tcp-common" module in terms of its features and groupings.

2.1.1. Model Scope

This document defines a common "grouping" statement for basic TCP connection parameters that matter to applications. In some TCP stacks, such parameters can also directly be set by an application using system calls, such as the socket API. The base YANG model in this document focuses on modeling TCP keep-alives. This base model can be extended as needed.

2.1.2. Features

The following diagram lists all the "feature" statements defined in the "ietf-tcp-common" module:

```
Features:
  +-- keepalives-supported
```

```
   |  The diagram above uses syntax that is similar to but not
   |  defined in [RFC8340].
```

2.1.3. Groupings

The following diagram lists all the "grouping" statements defined in the "ietf-keystore" module:

```
Groupings:
  +-- tcp-common-grouping
  +-- tcp-connection-grouping
```

> The diagram above uses syntax that is similar to but not defined in [RFC8340].

Each of these groupings are presented in the following subsections.

2.1.3.1. The "tcp-common-grouping" Grouping

The following tree diagram [RFC8340] illustrates the "tcp-common-grouping" grouping:

```
  grouping tcp-common-grouping
    +-- keepalives! {keepalives-supported}?
       +-- idle-time         uint16
       +-- max-probes        uint16
       +-- probe-interval    uint16
```

Comments:

* The "keepalives" node is a "presence" node so that the decendent nodes' "mandatory true" doesn't imply that keepalives must be configured.

* The "idle-time", "max-probes", and "probe-interval" nodes have the common meanings. Please see the YANG module in Section 2.3 for details.

2.1.3.2. The "tcp-connection-grouping" Grouping

The following tree diagram [RFC8340] illustrates the "tcp-connection-grouping" grouping:

```
  grouping tcp-connection-grouping
    +---u tcp-common-grouping
```

Comments:

* This grouping uses the "tcp-common-grouping" grouping discussed in Section 2.1.3.1.

2.1.4. Protocol-accessible Nodes

The "ietf-tcp-common" module does not contain any protocol-accessible nodes.

2.1.5. Guidelines for Configuring TCP Keep-Alives

Network stacks may include "keep-alives" in their TCP implementations, although this practice is not universally accepted. If keep-alives are included, [RFC1122] [RFC793bis] mandates that the application MUST be able to turn them on or off for each TCP connection, and that they MUST default to off.

Keep-alive mechanisms exist in many protocols. Depending on the protocol stack, TCP keep-alives may only be one out of several alternatives. Which mechanism(s) to use depends on the use case and application requirements. If keep-alives are needed by an application, it is RECOMMENDED that the aliveness check happens only at the protocol layers that are meaningful to the application.

A TCP keep-alive mechanism SHOULD only be invoked in server applications that might otherwise hang indefinitely and consume resources unnecessarily if a client crashes or aborts a connection during a network failure [RFC1122]. TCP keep-alives may consume significant resources both in the network and in endpoints (e.g., battery power). In addition, frequent keep-alives risk network congestion. The higher the frequency of keep-alives, the higher the overhead.

Given the cost of keep-alives, parameters have to be configured carefully:

* The default idle interval (leaf "idle-time") MUST default to no less than two hours, i.e., 7200 seconds [RFC1122]. A lower value MAY be configured, but keep-alive messages SHOULD NOT be transmitted more frequently than once every 15 seconds. Longer intervals SHOULD be used when possible.

* The maximum number of sequential keep-alive probes that can fail (leaf "max-probes") trades off responsiveness and robustness against packet loss. ACK segments that contain no data are not reliably transmitted by TCP. Consequently, if a keep-alive mechanism is implemented it MUST NOT interpret failure to respond to any specific probe as a dead connection [RFC1122]. Typically a single-digit number should suffice.

* TCP implementations may include a parameter for the number of seconds between TCP keep-alive probes (leaf "probe-interval"). In order to avoid congestion, the time interval between probes MUST NOT be smaller than one second. Significantly longer intervals SHOULD be used. It is important to note that keep-alive probes (or replies) can get dropped due to network congestion. Sending further probe messages into a congested path after a short interval, without backing off timers, could cause harm and result in a congestion collapse. Therefore it is essential to pick a large, conservative value for this interval.

2.2. Example Usage

This section presents an example showing the "tcp-common-grouping" populated with some data.

```
<tcp-common xmlns="urn:ietf:params:xml:ns:yang:ietf-tcp-common">
  <keepalives>
    <idle-time>15</idle-time>
    <max-probes>3</max-probes>
    <probe-interval>30</probe-interval>
  </keepalives>
</tcp-common>
```

2.3. YANG Module

The ietf-tcp-common YANG module references [RFC6991].

```
<CODE BEGINS> file "ietf-tcp-common@2020-08-20.yang"

module ietf-tcp-common {
  yang-version 1.1;
  namespace "urn:ietf:params:xml:ns:yang:ietf-tcp-common";
  prefix tcpcmn;

  organization
    "IETF NETCONF (Network Configuration) Working Group and the
     IETF TCP Maintenance and Minor Extensions (TCPM) Working Group";

  contact
    "WG Web:   <http://datatracker.ietf.org/wg/netconf/>
               <http://datatracker.ietf.org/wg/tcpm/>
     WG List:  <mailto:netconf@ietf.org>
               <mailto:tcpm@ietf.org>
     Authors:  Kent Watsen <mailto:kent+ietf@watsen.net>
               Michael Scharf
               <mailto:michael.scharf@hs-esslingen.de>";
```

```
  description
    "This module defines reusable groupings for TCP commons that
     can be used as a basis for specific TCP common instances.

     Copyright (c) 2020 IETF Trust and the persons identified
     as authors of the code. All rights reserved.

     Redistribution and use in source and binary forms, with
     or without modification, is permitted pursuant to, and
     subject to the license terms contained in, the Simplified
     BSD License set forth in Section 4.c of the IETF Trust's
     Legal Provisions Relating to IETF Documents
     (https://trustee.ietf.org/license-info).

     This version of this YANG module is part of RFC DDDD
     (https://www.rfc-editor.org/info/rfcDDDD); see the RFC
     itself for full legal notices.

     The key words 'MUST', 'MUST NOT', 'REQUIRED', 'SHALL',
     'SHALL NOT', 'SHOULD', 'SHOULD NOT', 'RECOMMENDED',
     'NOT RECOMMENDED', 'MAY', and 'OPTIONAL' in this document
     are to be interpreted as described in BCP 14 (RFC 2119)
     (RFC 8174) when, and only when, they appear in all
     capitals, as shown here.";

  revision 2020-08-20 {
    description
      "Initial version";
    reference
      "RFC DDDD: YANG Groupings for TCP Clients and TCP Servers";
  }

  // Features
  feature keepalives-supported {
    description
      "Indicates that keepalives are supported.";
  }

  // Groupings

  grouping tcp-common-grouping {
    description
      "A reusable grouping for configuring TCP parameters common
       to TCP connections as well as the operating system as a
       whole.";
    container keepalives {
      if-feature "keepalives-supported";
      presence
```

```
          "Indicates that keepalives are enabled.  Present so that
           the decendant nodes' 'mandatory true' doesn't imply that
           this node  must be configured.";
        description
          "Configures the keep-alive policy, to proactively test the
           aliveness of the TCP peer.  An unresponsive TCP peer is
           dropped after approximately (idle-time + max-probes
           * probe-interval) seconds.";
        leaf idle-time {
          type uint16 {
            range "1..max";
          }
          units "seconds";
          mandatory true;
          description
            "Sets the amount of time after which if no data has been
             received from the TCP peer, a TCP-level probe message
             will be sent to test the aliveness of the TCP peer.
             Two hours (7200 seconds) is safe value, per RFC 1122.";
          reference
            "RFC 1122:
              Requirements for Internet Hosts -- Communication Layers";
        }
        leaf max-probes {
          type uint16 {
            range "1..max";
          }
          mandatory true;
          description
            "Sets the maximum number of sequential keep-alive probes
             that can fail to obtain a response from the TCP peer
             before assuming the TCP peer is no longer alive.";
        }
        leaf probe-interval {
          type uint16 {
            range "1..max";
          }
          units "seconds";
          mandatory true;
          description
            "Sets the time interval between failed probes. The interval
             SHOULD be significantly longer than one second in order to
             avoid harm on a congested link.";
        }
      } // container keepalives
    } // grouping tcp-common-grouping
```

```
    grouping tcp-connection-grouping {
      description
        "A reusable grouping for configuring TCP parameters common
         to TCP connections.";
       uses tcp-common-grouping;
    }

  }
```

<CODE ENDS>

3. The "ietf-tcp-client" Module

This section defines a YANG 1.1 [RFC7950] module called "ietf-tcp-client". A high-level overview of the module is provided in Section 3.1. Examples illustatrating the module's use are provided in Examples (Section 3.2). The YANG module itself is defined in Section 3.3.

3.1. Data Model Overview

This section provides an overview of the "ietf-tcp-client" module in terms of its features and groupings.

3.1.1. Features

The following diagram lists all the "feature" statements defined in the "ietf-tcp-client" module:

```
Features:
  +-- local-binding-supported
  +-- tcp-client-keepalives
  +-- proxy-connect
  +-- socks5-gss-api
  +-- socks5-username-password
```

> The diagram above uses syntax that is similar to but not defined in [RFC8340].

3.1.2. Groupings

The following diagram lists all the "grouping" statements defined in the "ietf-tcp-client" module:

```
Groupings:
  +-- tcp-client-grouping
```

> The diagram above uses syntax that is similar to but not defined in [RFC8340].

Each of these groupings are presented in the following subsections.

3.1.2.1. The "tcp-client-grouping" Grouping

The following tree diagram [RFC8340] illustrates the "tcp-client-grouping" grouping:

```
  grouping tcp-client-grouping
    +-- remote-address                    inet:host
    +-- remote-port?                      inet:port-number
    +-- local-address?                    inet:ip-address
    |       {local-binding-supported}?
    +-- local-port?                       inet:port-number
    |       {local-binding-supported}?
    +-- proxy-server! {proxy-connect}?
    |  +-- (proxy-type)
    |     +--:(socks4)
    |     |  +-- socks4-parameters
    |     |     +-- remote-address    inet:ip-address
    |     |     +-- remote-port?      inet:port-number
    |     +--:(socks4a)
    |     |  +-- socks4a-parameters
    |     |     +-- remote-address    inet:host
    |     |     +-- remote-port?      inet:port-number
    |     +--:(socks5)
    |        +-- socks5-parameters
    |           +-- remote-address               inet:host
    |           +-- remote-port?                 inet:port-number
    |           +-- authentication-parameters!
    |              +-- (auth-type)
    |                 +--:(gss-api) {socks5-gss-api}?
    |                 |  +-- gss-api
    |                 +--:(username-password)
    |                           {socks5-username-password}?
    |                    +-- username-password
    |                       +-- username                string
    |                       +---u ct:password-grouping
    +---u tcpcmn:tcp-connection-grouping
```

Comments:

* The "remote-address" node, which is mandatory, may be configured as an IPv4 address, an IPv6 address, a hostname.

* The "remote-port" node is not mandatory, but its default value is the invalid value '0', thus forcing the consuming data model to refine it in order to provide it an appropriate default value.

* The "local-address" node, which is enabled by the "local-binding-supported" feature (Section 2.1.2), may be configured as an IPv4 address, an IPv6 address, or a wildcard value.

* The "local-port" node, which is enabled by the "local-binding-supported" feature (Section 2.1.2), is not mandatory. Its default value is '0', indicating that the operating system can pick an arbitrary port number.

* The "proxy-server" node is enabled by a "feature" statement and, for servers that enable it, is a "presence" container so that the decendent "mandatory true" choice node doesn't imply that the proxt-server node must be configured.

* This grouping uses the "tcp-connection-grouping" grouping discussed in Section 2.1.3.2.

3.1.3. Protocol-accessible Nodes

The "ietf-tcp-client" module does not contain any protocol-accessible nodes.

3.2. Example Usage

This section presents two examples showing the "tcp-client-grouping" populated with some data. This example shows a TCP-client configured to not connect via a proxy:

```
<tcp-client xmlns="urn:ietf:params:xml:ns:yang:ietf-tcp-client">
  <remote-address>www.example.com</remote-address>
  <remote-port>443</remote-port>
  <local-address>0.0.0.0</local-address>
  <local-port>0</local-port>
  <keepalives>
    <idle-time>15</idle-time>
    <max-probes>3</max-probes>
    <probe-interval>30</probe-interval>
  </keepalives>
</tcp-client>
```

This example shows a TCP-client configured to connect via a proxy:

```
<tcp-client xmlns="urn:ietf:params:xml:ns:yang:ietf-tcp-client">
  <remote-address>www.example.com</remote-address>
  <remote-port>443</remote-port>
  <local-address>0.0.0.0</local-address>
  <local-port>0</local-port>
  <proxy-server>
    <socks5-parameters>
      <remote-address>proxy.my-domain.com</remote-address>
      <remote-port>1080</remote-port>
      <authentication-parameters>
        <username-password>
          <username>foobar</username>
          <cleartext-password>secret</cleartext-password>
        </username-password>
      </authentication-parameters>
    </socks5-parameters>
  </proxy-server>
  <keepalives>
    <idle-time>15</idle-time>
    <max-probes>3</max-probes>
    <probe-interval>30</probe-interval>
  </keepalives>
</tcp-client>
```

3.3. YANG Module

The ietf-tcp-client YANG module references [RFC6991].

```
<CODE BEGINS> file "ietf-tcp-client@2020-08-20.yang"

module ietf-tcp-client {
  yang-version 1.1;
  namespace "urn:ietf:params:xml:ns:yang:ietf-tcp-client";
  prefix tcpc;

  import ietf-inet-types {
    prefix inet;
    reference
      "RFC 6991: Common YANG Data Types";
  }

  import ietf-crypto-types {
    prefix ct;
    reference
      "RFC AAAA: YANG Data Types and Groupings for Cryptography";
  }

  import ietf-tcp-common {
```

```
      prefix tcpcmn;
      reference
        "RFC DDDD: YANG Groupings for TCP Clients and TCP Servers";
    }

    organization
      "IETF NETCONF (Network Configuration) Working Group and the
       IETF TCP Maintenance and Minor Extensions (TCPM) Working Group";

    contact
      "WG Web:   <http://datatracker.ietf.org/wg/netconf/>
                 <http://datatracker.ietf.org/wg/tcpm/>
       WG List:  <mailto:netconf@ietf.org>
                 <mailto:tcpm@ietf.org>
       Authors:  Kent Watsen <mailto:kent+ietf@watsen.net>
                 Michael Scharf
                 <mailto:michael.scharf@hs-esslingen.de>";

    description
      "This module defines reusable groupings for TCP clients that
       can be used as a basis for specific TCP client instances.

       Copyright (c) 2020 IETF Trust and the persons identified
       as authors of the code. All rights reserved.

       Redistribution and use in source and binary forms, with
       or without modification, is permitted pursuant to, and
       subject to the license terms contained in, the Simplified
       BSD License set forth in Section 4.c of the IETF Trust's
       Legal Provisions Relating to IETF Documents
       (https://trustee.ietf.org/license-info).

       This version of this YANG module is part of RFC DDDD
       (https://www.rfc-editor.org/info/rfcDDDD); see the RFC
       itself for full legal notices.

       The key words 'MUST', 'MUST NOT', 'REQUIRED', 'SHALL',
       'SHALL NOT', 'SHOULD', 'SHOULD NOT', 'RECOMMENDED',
       'NOT RECOMMENDED', 'MAY', and 'OPTIONAL' in this document
       are to be interpreted as described in BCP 14 (RFC 2119)
       (RFC 8174) when, and only when, they appear in all
       capitals, as shown here.";

    revision 2020-08-20 {
      description
        "Initial version";
      reference
        "RFC DDDD: YANG Groupings for TCP Clients and TCP Servers";
```

```
  }

  // Features

  feature local-binding-supported {
    description
      "Indicates that the server supports configuring local
       bindings (i.e., the local address and local port) for
       TCP clients.";
  }

  feature tcp-client-keepalives {
    description
      "Per socket TCP keepalive parameters are configurable for
       TCP clients on the server implementing this feature.";
  }

  feature proxy-connect {
    description
      "Proxy connection configuration is configurable for
       TCP clients on the server implementing this feature.";
  }

  feature socks5-gss-api {
    description
      "Indicates that the server supports authenticating
       using GSSAPI when initiating TCP connections via
       and SOCKS Version 5 proxy server.";
    reference
      "RFC 1928: SOCKS Protocol Version 5";
  }

  feature socks5-username-password {
    description
      "Indicates that the server supports authenticating
       using username/password when initiating TCP
       connections via and SOCKS Version 5 proxy
       server.";
    reference
      "RFC 1928: SOCKS Protocol Version 5";
  }

  // Groupings

  grouping tcp-client-grouping {
    description
      "A reusable grouping for configuring a TCP client.
```

```
          Note that this grouping uses fairly typical descendent
           node names such that a stack of 'uses' statements will
           have name conflicts.  It is intended that the consuming
           data model will resolve the issue (e.g., by wrapping
           the 'uses' statement in a container called
           'tcp-client-parameters').  This model purposely does
           not do this itself so as to provide maximum flexibility
           to consuming models.";

        leaf remote-address {
          type inet:host;
          mandatory true;
          description
            "The IP address or hostname of the remote peer to
             establish a connection with.  If a domain name is
             configured, then the DNS resolution should happen on
             each connection attempt.  If the DNS resolution
             results in multiple IP addresses, the IP addresses
             are tried according to local preference order until
             a connection has been established or until all IP
             addresses have failed.";
        }
        leaf remote-port {
          type inet:port-number;
          default "0";
          description
            "The IP port number for the remote peer to establish a
             connection with.  An invalid default value (0) is used
             (instead of 'mandatory true') so that as application
             level data model may 'refine' it with an application
             specific default port number value.";
        }
        leaf local-address {
          if-feature "local-binding-supported";
          type inet:ip-address;
          description
            "The local IP address/interface (VRF?) to bind to for when
             connecting to the remote peer.  INADDR_ANY ('0.0.0.0') or
             INADDR6_ANY ('0:0:0:0:0:0:0:0' a.k.a. '::') MAY be used to
             explicitly indicate the implicit default, that the server
             can bind to any IPv4 or IPv6 addresses, respectively.";
        }
        leaf local-port {
          if-feature "local-binding-supported";
          type inet:port-number;
          default "0";
          description
            "The local IP port number to bind to for when connecting
```

```
            to the remote peer.  The port number '0', which is the
            default value, indicates that any available local port
            number may be used.";
        }

        container proxy-server {
          if-feature "proxy-connect";
          presence
            "Indicates that a proxy connection is configured.
             Present so that the 'proxy-type' node's 'mandatory
             true' doesn't imply that the proxy connection
             must be configured.";
          choice proxy-type {
            mandatory true;
            description
              "Selects a proxy connection protocol.";
            case socks4 {
              container socks4-parameters {
                leaf remote-address {
                  type inet:ip-address;
                  mandatory true;
                  description
                    "The IP address of the proxy server.";
                }
                leaf remote-port {
                  type inet:port-number;
                  default "1080";
                  description
                    "The IP port number for the proxy server.";
                }
                description
                  "Parameters for connecting to a TCP-based proxy
                   server using the SOCKS4 protocol.";
                reference
                  "SOCKS, Proceedings: 1992 Usenix Security Symposium.";
              }
            }
            case socks4a {
              container socks4a-parameters {
                leaf remote-address {
                  type inet:host;
                  mandatory true;
                  description
                    "The IP address or hostname of the proxy server.";
                }
                leaf remote-port {
                  type inet:port-number;
                  default "1080";
```

```
            description
              "The IP port number for the proxy server.";
          }
          description
            "Parameters for connecting to a TCP-based proxy
             server using the SOCKS4a protocol.";
          reference
            "SOCKS Proceedings:
               1992 Usenix Security Symposium.
             OpenSSH message:
               SOCKS 4A: A Simple Extension to SOCKS 4 Protocol
               https://www.openssh.com/txt/socks4a.protocol";
        }
      }
      case socks5 {
        container socks5-parameters {
          leaf remote-address {
            type inet:host;
            mandatory true;
            description
              "The IP address or hostname of the proxy server.";
          }
          leaf remote-port {
            type inet:port-number;
            default "1080";
            description
              "The IP port number for the proxy server.";
          }
          container authentication-parameters {
            presence
              "Indicates that an authentication mechanism
               has been configured.  Present so that the
               'auth-type' node's 'mandatory true' doesn't
               imply that an authentication mechanism
               must be configured.";
            description
              "A container for SOCKS Version 5 authentication
               mechanisms.

               A complete list of methods is defined at:
               https://www.iana.org/assignments/socks-methods
               /socks-methods.xhtml.";
            reference
              "RFC 1928: SOCKS Protocol Version 5";
            choice auth-type {
              mandatory true;
              description
                "A choice amongst supported SOCKS Version 5
```

```
                   authentication mechanisms.";
                 case gss-api {
                   if-feature socks5-gss-api;
                   container gss-api {
                     description
                       "Contains GSS-API configuration.  Defines
                        as an empty container to enable specific
                        GSS-API configuration to be augmented in
                        by future modules.";
                     reference
                       "RFC 1928: SOCKS Protocol Version 5
                        RFC 2743: Generic Security Service
                                  Application Program Interface
                                  Version 2, Update 1";
                   }
                 }
                 case username-password {
                   if-feature socks5-username-password;
                   container username-password {
                     leaf username {
                       type string;
                       mandatory true;
                       description
                         "The 'username' value to use for client
                          identification.";
                     }
                     uses ct:password-grouping {
                       description
                         "The password to be used for client
                          authentication.";
                     }
                     description
                       "Contains Username/Password configuration.";
                     reference
                       "RFC 1929: Username/Password Authentication
                                  for SOCKS V5";
                   }
                 }
               }
             }
             description
               "Parameters for connecting to a TCP-based proxy server
                using the SOCKS5 protocol.";
             reference
               "RFC 1928: SOCKS Protocol Version 5";
           }
         }
       }
```

```
          description
            "Proxy server settings.";
        }

        uses tcpcmn:tcp-connection-grouping {
          augment "keepalives" {
            if-feature "tcp-client-keepalives";
            description
              "Add an if-feature statement so that implementations
               can choose to support TCP client keepalives.";
          }
        }
      }
    }
```

<CODE ENDS>

4. The "ietf-tcp-server" Module

This section defines a YANG 1.1 [RFC7950] module called "ietf-tcp-server". A high-level overview of the module is provided in Section 4.1. Examples illustatrating the module's use are provided in Examples (Section 4.2). The YANG module itself is defined in Section 4.3.

4.1. Data Model Overview

This section provides an overview of the "ietf-tcp-server" module in terms of its features and groupings.

4.1.1. Features

The following diagram lists all the "feature" statements defined in the "ietf-tcp-server" module:

```
Features:
  +-- tcp-server-keepalives
```

> The diagram above uses syntax that is similar to but not defined in [RFC8340].

4.1.2. Groupings

The following diagram lists all the "grouping" statements defined in the "ietf-tcp-server" module:

```
Groupings:
  +-- tcp-server-grouping
```

> The diagram above uses syntax that is similar to but not defined in [RFC8340].

Each of these groupings are presented in the following subsections.

4.1.2.1. The "tcp-server-grouping" Grouping

The following tree diagram [RFC8340] illustrates the "tcp-server-grouping" grouping:

```
  grouping tcp-server-grouping
    +-- local-address                      inet:ip-address
    +-- local-port?                        inet:port-number
    +---u tcpcmn:tcp-connection-grouping
```

Comments:

* The "local-address" node, which is mandatory, may be configured as an IPv4 address, an IPv6 address, or a wildcard value.

* The "local-port" node is not mandatory, but its default value is the invalid value '0', thus forcing the consuming data model to refine it in order to provide it an appropriate default value.

* This grouping uses the "tcp-connection-grouping" grouping discussed in Section 2.1.3.2.

4.1.3. Protocol-accessible Nodes

The "ietf-tcp-server" module does not contain any protocol-accessible nodes.

4.2. Example Usage

This section presents an example showing the "tcp-server-grouping" populated with some data.

```
<tcp-server xmlns="urn:ietf:params:xml:ns:yang:ietf-tcp-server">
  <local-address>10.20.30.40</local-address>
  <local-port>7777</local-port>
  <keepalives>
    <idle-time>15</idle-time>
    <max-probes>3</max-probes>
    <probe-interval>30</probe-interval>
  </keepalives>
</tcp-server>
```

4.3. YANG Module

The ietf-tcp-server YANG module references [RFC6991].

<CODE BEGINS> file "ietf-tcp-server@2020-08-20.yang"

```
module ietf-tcp-server {
  yang-version 1.1;
  namespace "urn:ietf:params:xml:ns:yang:ietf-tcp-server";
  prefix tcps;

  import ietf-inet-types {
    prefix inet;
    reference
      "RFC 6991: Common YANG Data Types";
  }

  import ietf-tcp-common {
    prefix tcpcmn;
    reference
      "RFC DDDD: YANG Groupings for TCP Clients and TCP Servers";
  }

  organization
    "IETF NETCONF (Network Configuration) Working Group and the
     IETF TCP Maintenance and Minor Extensions (TCPM) Working Group";

  contact
    "WG Web:   <http://datatracker.ietf.org/wg/netconf/>
               <http://datatracker.ietf.org/wg/tcpm/>
     WG List:  <mailto:netconf@ietf.org>
               <mailto:tcpm@ietf.org>
     Authors:  Kent Watsen <mailto:kent+ietf@watsen.net>
               Michael Scharf
               <mailto:michael.scharf@hs-esslingen.de>";

  description
    "This module defines reusable groupings for TCP servers that
     can be used as a basis for specific TCP server instances.

     Copyright (c) 2020 IETF Trust and the persons identified
     as authors of the code. All rights reserved.

     Redistribution and use in source and binary forms, with
     or without modification, is permitted pursuant to, and
     subject to the license terms contained in, the Simplified
     BSD License set forth in Section 4.c of the IETF Trust's
     Legal Provisions Relating to IETF Documents
```

```
        (https://trustee.ietf.org/license-info).

        This version of this YANG module is part of RFC DDDD
        (https://www.rfc-editor.org/info/rfcDDDD); see the RFC
        itself for full legal notices.

        The key words 'MUST', 'MUST NOT', 'REQUIRED', 'SHALL',
        'SHALL NOT', 'SHOULD', 'SHOULD NOT', 'RECOMMENDED',
        'NOT RECOMMENDED', 'MAY', and 'OPTIONAL' in this document
        are to be interpreted as described in BCP 14 (RFC 2119)
        (RFC 8174) when, and only when, they appear in all
        capitals, as shown here.";

    revision 2020-08-20 {
      description
        "Initial version";
      reference
        "RFC DDDD: YANG Groupings for TCP Clients and TCP Servers";
    }

    // Features

    feature tcp-server-keepalives {
      description
        "Per socket TCP keepalive parameters are configurable for
         TCP servers on the server implementing this feature.";
    }


    // Groupings

    grouping tcp-server-grouping {
      description
        "A reusable grouping for configuring a TCP server.

         Note that this grouping uses fairly typical descendent
         node names such that a stack of 'uses' statements will
         have name conflicts.  It is intended that the consuming
         data model will resolve the issue (e.g., by wrapping
         the 'uses' statement in a container called
         'tcp-server-parameters').  This model purposely does
         not do this itself so as to provide maximum flexibility
         to consuming models.";
      leaf local-address {
        type inet:ip-address;
        mandatory true;
        description
          "The local IP address to listen on for incoming
```

```
            TCP client connections.  INADDR_ANY (0.0.0.0) or
            INADDR6_ANY (0:0:0:0:0:0:0:0 a.k.a. ::) MUST be
            used when the server is to listen on all IPv4 or
            IPv6 addresses, respectively.";
        }
        leaf local-port {
          type inet:port-number;
          default "0";
          description
            "The local port number to listen on for incoming TCP
             client connections.  An invalid default value (0)
             is used (instead of 'mandatory true') so that an
             application level data model may 'refine' it with
             an application specific default port number value.";
        }
        uses tcpcmn:tcp-connection-grouping {
          augment "keepalives" {
            if-feature "tcp-server-keepalives";
            description
              "Add an if-feature statement so that implementations
               can choose to support TCP server keepalives.";
          }
        }
      }
    }

    <CODE ENDS>
```

5. Security Considerations

5.1. The "ietf-tcp-common" YANG Module

The "ietf-tcp-common" YANG module defines "grouping" statements that are designed to be accessed via YANG based management protocols, such as NETCONF [RFC6241] and RESTCONF [RFC8040]. Both of these protocols have mandatory-to-implement secure transport layers (e.g., SSH, TLS) with mutual authentication.

The NETCONF access control model (NACM) [RFC8341] provides the means to restrict access for particular users to a pre-configured subset of all available protocol operations and content.

Since the module in this document only define groupings, these considerations are primarily for the designers of other modules that use these groupings.

None of the readable data nodes defined in this YANG module are considered sensitive or vulnerable in network environments. The NACM "default-deny-all" extension has not been set for any data nodes defined in this module.

None of the writable data nodes defined in this YANG module are considered sensitive or vulnerable in network environments. The NACM "default-deny-write" extension has not been set for any data nodes defined in this module.

This module does not define any RPCs, actions, or notifications, and thus the security consideration for such is not provided here.

5.2. The "ietf-tcp-client" YANG Module

The "ietf-tcp-client" YANG module defines "grouping" statements that are designed to be accessed via YANG based management protocols, such as NETCONF [RFC6241] and RESTCONF [RFC8040]. Both of these protocols have mandatory-to-implement secure transport layers (e.g., SSH, TLS) with mutual authentication.

The NETCONF access control model (NACM) [RFC8341] provides the means to restrict access for particular users to a pre-configured subset of all available protocol operations and content.

Since the module in this document only define groupings, these considerations are primarily for the designers of other modules that use these groupings.

One readable data node defined in this YANG module may be considered sensitive or vulnerable in some network environments. This node is as follows:

* The "proxy-server/socks5-parameters/authentication-parameters/username-password/password" node:

  The cleartext "password" node defined in the "tcp-client-grouping" grouping is additionally sensitive to read operations such that, in normal use cases, it should never be returned to a client. For this reason, the NACM extension "default-deny-all" has been applied to it.

None of the writable data nodes defined in this YANG module are considered sensitive or vulnerable in network environments. The NACM "default-deny-write" extension has not been set for any data nodes defined in this module.

This module does not define any RPCs, actions, or notifications, and thus the security consideration for such is not provided here.

5.3. The "ietf-tcp-server" YANG Module

The "ietf-tcp-server" YANG module defines "grouping" statements that are designed to be accessed via YANG based management protocols, such as NETCONF [RFC6241] and RESTCONF [RFC8040]. Both of these protocols have mandatory-to-implement secure transport layers (e.g., SSH, TLS) with mutual authentication.

The NETCONF access control model (NACM) [RFC8341] provides the means to restrict access for particular users to a pre-configured subset of all available protocol operations and content.

Since the module in this document only define groupings, these considerations are primarily for the designers of other modules that use these groupings.

None of the readable data nodes defined in this YANG module are considered sensitive or vulnerable in network environments. The NACM "default-deny-all" extension has not been set for any data nodes defined in this module.

None of the writable data nodes defined in this YANG module are considered sensitive or vulnerable in network environments. The NACM "default-deny-write" extension has not been set for any data nodes defined in this module.

This module does not define any RPCs, actions, or notifications, and thus the security consideration for such is not provided here.

6. IANA Considerations

6.1. The "IETF XML" Registry

This document registers two URIs in the "ns" subregistry of the IETF XML Registry [RFC3688]. Following the format in [RFC3688], the following registrations are requested:

```
   URI: urn:ietf:params:xml:ns:yang:ietf-tcp-common
   Registrant Contact: The NETCONF WG of the IETF.
   XML: N/A, the requested URI is an XML namespace.

   URI: urn:ietf:params:xml:ns:yang:ietf-tcp-client
   Registrant Contact: The NETCONF WG of the IETF.
   XML: N/A, the requested URI is an XML namespace.

   URI: urn:ietf:params:xml:ns:yang:ietf-tcp-server
   Registrant Contact: The NETCONF WG of the IETF.
   XML: N/A, the requested URI is an XML namespace.
```

6.2. The "YANG Module Names" Registry

This document registers two YANG modules in the YANG Module Names registry [RFC6020]. Following the format in [RFC6020], the following registrations are requested:

```
   name:         ietf-tcp-common
   namespace:    urn:ietf:params:xml:ns:yang:ietf-tcp-common
   prefix:       tcpcmn
   reference:    RFC DDDD

   name:         ietf-tcp-client
   namespace:    urn:ietf:params:xml:ns:yang:ietf-tcp-client
   prefix:       tcpc
   reference:    RFC DDDD

   name:         ietf-tcp-server
   namespace:    urn:ietf:params:xml:ns:yang:ietf-tcp-server
   prefix:       tcps
   reference:    RFC DDDD
```

7. References

7.1. Normative References

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

[RFC6020] Bjorklund, M., Ed., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020, DOI 10.17487/RFC6020, October 2010, <https://www.rfc-editor.org/info/rfc6020>.

[RFC6991] Schoenwaelder, J., Ed., "Common YANG Data Types", RFC 6991, DOI 10.17487/RFC6991, July 2013, <https://www.rfc-editor.org/info/rfc6991>.

[RFC7950] Bjorklund, M., Ed., "The YANG 1.1 Data Modeling Language", RFC 7950, DOI 10.17487/RFC7950, August 2016, <https://www.rfc-editor.org/info/rfc7950>.

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

[RFC8341] Bierman, A. and M. Bjorklund, "Network Configuration Access Control Model", STD 91, RFC 8341, DOI 10.17487/RFC8341, March 2018, <https://www.rfc-editor.org/info/rfc8341>.

7.2. Informative References

[I-D.ietf-netconf-crypto-types]
Watsen, K., "YANG Data Types and Groupings for Cryptography", Work in Progress, Internet-Draft, draft-ietf-netconf-crypto-types-17, 10 July 2020, <https://tools.ietf.org/html/draft-ietf-netconf-crypto-types-17>.

[I-D.ietf-netconf-http-client-server]
Watsen, K., "YANG Groupings for HTTP Clients and HTTP Servers", Work in Progress, Internet-Draft, draft-ietf-netconf-http-client-server-04, 8 July 2020, <https://tools.ietf.org/html/draft-ietf-netconf-http-client-server-04>.

[I-D.ietf-netconf-keystore]
Watsen, K., "A YANG Data Model for a Keystore", Work in Progress, Internet-Draft, draft-ietf-netconf-keystore-19, 10 July 2020, <https://tools.ietf.org/html/draft-ietf-netconf-keystore-19>.

[I-D.ietf-netconf-netconf-client-server]
Watsen, K., "NETCONF Client and Server Models", Work in Progress, Internet-Draft, draft-ietf-netconf-netconf-client-server-20, 8 July 2020, <https://tools.ietf.org/html/draft-ietf-netconf-netconf-client-server-20>.

[I-D.ietf-netconf-restconf-client-server]
Watsen, K., "RESTCONF Client and Server Models", Work in Progress, Internet-Draft, draft-ietf-netconf-restconf-client-server-20, 8 July 2020, <https://tools.ietf.org/html/draft-ietf-netconf-restconf-client-server-20>.

[I-D.ietf-netconf-ssh-client-server]
Watsen, K. and G. Wu, "YANG Groupings for SSH Clients and SSH Servers", Work in Progress, Internet-Draft, draft-ietf-netconf-ssh-client-server-21, 10 July 2020, <https://tools.ietf.org/html/draft-ietf-netconf-ssh-client-server-21>.

[I-D.ietf-netconf-tcp-client-server]
Watsen, K. and M. Scharf, "YANG Groupings for TCP Clients and TCP Servers", Work in Progress, Internet-Draft, draft-ietf-netconf-tcp-client-server-07, 8 July 2020, <https://tools.ietf.org/html/draft-ietf-netconf-tcp-client-server-07>.

[I-D.ietf-netconf-tls-client-server]
Watsen, K. and G. Wu, "YANG Groupings for TLS Clients and TLS Servers", Work in Progress, Internet-Draft, draft-ietf-netconf-tls-client-server-21, 10 July 2020, <https://tools.ietf.org/html/draft-ietf-netconf-tls-client-server-21>.

[I-D.ietf-netconf-trust-anchors]
Watsen, K., "A YANG Data Model for a Truststore", Work in Progress, Internet-Draft, draft-ietf-netconf-trust-anchors-12, 10 July 2020, <https://tools.ietf.org/html/draft-ietf-netconf-trust-anchors-12>.

[RFC3688] Mealling, M., "The IETF XML Registry", BCP 81, RFC 3688, DOI 10.17487/RFC3688, January 2004, <https://www.rfc-editor.org/info/rfc3688>.

[RFC6241] Enns, R., Ed., Bjorklund, M., Ed., Schoenwaelder, J., Ed., and A. Bierman, Ed., "Network Configuration Protocol (NETCONF)", RFC 6241, DOI 10.17487/RFC6241, June 2011, <https://www.rfc-editor.org/info/rfc6241>.

[RFC8040] Bierman, A., Bjorklund, M., and K. Watsen, "RESTCONF Protocol", RFC 8040, DOI 10.17487/RFC8040, January 2017, <https://www.rfc-editor.org/info/rfc8040>.

[RFC8340] Bjorklund, M. and L. Berger, Ed., "YANG Tree Diagrams", BCP 215, RFC 8340, DOI 10.17487/RFC8340, March 2018, <https://www.rfc-editor.org/info/rfc8340>.

[RFC8342] Bjorklund, M., Schoenwaelder, J., Shafer, P., Watsen, K., and R. Wilton, "Network Management Datastore Architecture (NMDA)", RFC 8342, DOI 10.17487/RFC8342, March 2018, <https://www.rfc-editor.org/info/rfc8342>.

Appendix A. Change Log

This section is to be removed before publishing as an RFC.

A.1. 00 to 01

* Added 'local-binding-supported' feature to TCP-client model.

* Added 'keepalives-supported' feature to TCP-common model.

* Added 'external-endpoint-values' container and 'external-endpoints' feature to TCP-server model.

A.2. 01 to 02

* Removed the 'external-endpoint-values' container and 'external-endpoints' feature from the TCP-server model.

A.3. 02 to 03

* Moved the common model section to be before the client and server specific sections.

* Added sections "Model Scope" and "Usage Guidelines for Configuring TCP Keep-Alives" to the common model section.

A.4. 03 to 04

* Fixed a few typos.

A.5. 04 to 05

* Removed commented out "grouping tcp-system-grouping" statement kept for reviewers.

* Added a "Note to Reviewers" note to first page.

A.6. 05 to 06

* Added support for TCP proxies.

A.7. 06 to 07

* Expanded "Data Model Overview section(s) [remove "wall" of tree diagrams].

* Updated the Security Considerations section.

A.8. 08 to 09

* Added missing IANA registration for "ietf-tcp-common"

* Added "mandatory true" for the "username" and "password" leafs

* Added an example of a TCP-client configured to connect via a proxy

* Fixed issues found by the SecDir review of the "keystore" draft.

* Updated the "ietf-tcp-client" module to use the new "password-grouping" grouping from the "crypto-types" module.

Authors' Addresses

Kent Watsen
Watsen Networks

Email: kent+ietf@watsen.net

Michael Scharf
Hochschule Esslingen - University of Applied Sciences

Email: michael.scharf@hs-esslingen.de

TCPM WG
Internet Draft
Intended status: Informational
Obsoletes: 2140
Expires: June 2021

J. Touch
Independent
M. Welzl
S. Islam
University of Oslo

December 28, 2020

TCP Control Block Interdependence
draft-ietf-tcpm-2140bis-07.txt

Status of this Memo

Abstract

This memo provides guidance to TCP implementers that are intended to help improve convergence to steady-state operation without affecting interoperability. It updates and replaces RFC 2140's description of interdependent TCP control blocks and the ways that part of TCP state can be shared among similar concurrent or consecutive connections. TCP state includes a combination of parameters, such as connection state, current round-trip time estimates, congestion control information, and process information. Most of this state is maintained on a per-connection basis in the TCP Control Block (TCB), but implementations can (and do) share certain TCB information across connections to the same host. Such sharing is intended to improve overall transient transport performance, while maintaining backward-compatibility with existing implementations. The sharing described herein is limited to only the TCB initialization and so has no effect on the long-term behavior of TCP after a connection has been established.

Table of Contents

1. Introduction

TCP is a connection-oriented reliable transport protocol layered over IP [RFC793]. Each TCP connection maintains state, usually in a data structure called the TCP Control Block (TCB). The TCB contains information about the connection state, its associated local process, and feedback parameters about the connection's transmission properties. As originally specified and usually implemented, most TCB information is maintained on a per-connection basis. Some implementations can (and now do) share certain TCB information across connections to the same host [RFC2140]. Such sharing is intended to lead to better overall transient performance, especially for numerous short-lived and simultaneous connections, as often used in the World-Wide Web [Be94][Br02]. This sharing of state is intended to help TCP connections converge to steady-state behavior more quickly without affecting TCP interoperability.

This document updates RFC 2140's discussion of TCB state sharing and provides a complete replacement for that document. This state sharing affects only TCB initialization [RFC2140] and thus has no effect on the long-term behavior of TCP after a connection has been established nor on interoperability. Path information shared across SYN destination port numbers assumes that TCP segments having the same host-pair experience the same path properties, irrespective of TCP port numbers. The observations about TCB sharing in this document apply similarly to any protocol with congestion state, including SCTP [RFC4960] and DCCP [RFC4340], as well as for individual subflows in Multipath TCP [RFC8684].

2. Conventions Used in This Document

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

The core of this document describes behavior that is already permitted by TCP standards. As a result, it provides informative guidance but does not use normative language, except when quoting other documents. Normative language is used in Appendix C as examples of requirements for future consideration.

3. Terminology

The following terminology is used frequently in this document. Items preceded with a "+" may be part of the state maintained as TCP connection state in the associated connections TCB and are the focus of sharing as described in this document.

+cwnd - the TCP congestion window size [RFC5681]

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

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

+MMS_R - the maximum message size that can be received, the largest received transport payload of an IP datagram [RFC1122]

+MMS_S - the maximum message size that can be sent, the largest transmitted transport payload of an IP datagram [RFC1122]

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

PCB - protocol control block, the data associated with a protocol as maintained by an endpoint; a TCP PCB is called a TCB

PLPMTUD - packetization-layer path MTU discovery, a mechanism that uses transport packets to discovery the PMTU [RFC4821]

+PMTU - the largest IP datagram that can traverse a path [RFC1191][RFC8201]

PMTUD - path-layer MTU discovery, a mechanism that relies on ICMP error messages to discover the PMTU [RFC1191][RFC8201]

+RTT - the round-trip time of a TCP packet exchange [RFC793]

+RTTvar - the variance of the round-trip times of a TCP packet exchange [RFC6298]

+RWIN - the TCP receive window size [RFC793]

+sendcwnd - the TCP send-side congestion window (cwnd) size [RFC5681]

+sendMSS - the TCP maximum segment size, a value transmitted in a TCP option that represents the largest TCP user data payload that can be received [RFC793]

+ssthresh - the TCP slow-start threshold [RFC5681]

TCB - TCP Control Block, the data associated with a TCP connection as maintained by an endpoint

TCP-AO - the TCP Authentication Option [RFC5925]

TFO - TCP Fast Open option [RFC7413]

+TFO_cookie - the TCP Fast Open cookie, state that is used as part of the TFO mechanism, when TFO is supported [RFC7413]

+TFO_failure - an indication of when TFO option negotiation failed, when TFO is supported

+TFOinfo - information cached when a TFO connection is established, which includes the TFO_cookie [RFC7413]

4. The TCP Control Block (TCB)

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

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

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

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

5. TCB Interdependence

There are two cases of TCB interdependence. Temporal sharing occurs when the TCB of an earlier (now CLOSED) connection to a host is used

to initialize some parameters of a new connection to that same host, i.e., in sequence. Ensemble sharing occurs when a currently active connection to a host is used to initialize another (concurrent) connection to that host.

6. Temporal Sharing

The TCB data cache is accessed in two ways: it is read to initialize new TCBs and written when more current per-host state is available.

6.1. Initialization of the new TCB

TCBs for new connections can be initialized using context from past connections as follows:

TEMPORAL SHARING - TCB Initialization

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

The table below gives an overview of option-specific information that can be shared. Additional information on some specific TCP options and sharing is provided in Appendix B.

TEMPORAL SHARING - Option Info Initialization

Cached	New
old_TFO_cookie	old_TFO_cookie
old_TFO_failure	old_TFO_failure

6.2. Updates to the new TCB

During the connection, the associated TCB can be updated based on particular events, as shown below:

TEMPORAL SHARING - Cache Updates

Cached TCB	Current TCB	when?	New Cached TCB
old_MMS_S	curr_MMS_S	OPEN	curr_MMS_S
old_MMS_R	curr_MMS_R	OPEN	curr_MMS_R
old_sendMSS	curr_sendMSS	MSSopt	curr_sendMSS
old_PMTU	curr_PMTU	PMTUD	curr_PMTU
old_RTT	curr_RTT	CLOSE	merge(curr,old)
old_RTTvar	curr_RTTvar	CLOSE	merge(curr,old)
old_option	curr_option	ESTAB	(depends on option)
old_ssthresh	curr_ssthresh	CLOSE	merge(curr,old)
old_sendcwnd	curr_sendcwnd	CLOSE	merge(curr,old)

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

TEMPORAL SHARING - Option Info Updates

Cached	Current	when?	New Cached
old_TFO_cookie	old_TFO_cookie	ESTAB	old_TFO_cookie
old_TFO_failure	old_TFO_failure	ESTAB	old_TFO_failure

6.3. Discussion

There is no particular benefit to caching MMS_S and MMS_R as these are reported by the local IP stack. Caching sendMSS and PMTU is trivial; reported values are cached, and the most recent values are used. The cache is updated when the MSS option is received in a SYN or after PMTUD (i.e., when an ICMPv4 Fraqmentation Needed [RFC1191] or ICMPv6 Packet Too Big message is received [RFC8201] or the equivalent is inferred, e.g., as from PLPMTUD [RFC4821]), respectively, so the cache always has the most recent values from any connection. For sendMSS, the cache is consulted only at connection establishment and not otherwise updated, which means that MSS options do not affect current connections. The default sendMSS is never saved; only reported MSS values update the cache, so an explicit override is required to reduce the sendMSS.

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

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

TCP options are copied or merged depending on the details of each option, where "merge" is some function that combines the values of "curr" and "old". E.g., TFO state is updated when a connection is established and read before establishing a new connection.

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

Most cached TCB values are updated when a connection closes. The exceptions are MMS_R and MMS_S, which are reported by IP [RFC1122], PMTU which is updated after Path MTU Discovery [RFC1191][RFC4821][RFC8201], and sendMSS, which is updated if the MSS option is received in the TCP SYN header.

Sharing sendMSS information affects only data in the SYN of the next connection, because sendMSS information is typically included in most TCP SYN segments. Caching PMTU can accelerate the efficiency of

PMTUD but can also result in black-holing until corrected if in error. Caching MMS_R and MMS_S may be of little direct value as they are reported by the local IP stack anyway.

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

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

7. Ensemble Sharing

Sharing cached TCB data across concurrent connections requires attention to the aggregate nature of some of the shared state. For example, although MSS and RTT values can be shared by copying, it may not be appropriate to simply copy congestion window or ssthresh information; instead, the new values can be a function (f) of the cumulative values and the number of connections (N).

7.1. Initialization of a new TCB

TCBs for new connections can be initialized using context from concurrent connections as follows:

ENSEMBLE SHARING - TCB Initialization

Cached TCB	New TCB
old_MMS_S	old_MMS_S
old_MMS_R	old_MMS_R
old_sendMSS	old_sendMSS
old_PMTU	old_PMTU
old_RTT	old_RTT
old_RTTvar	old_RTTvar
sum(old_ssthresh)	f(sum(old_ssthresh), N)
sum(old_sendcwnd)	f(sum(old_sendcwnd), N)
old_option	(option specific)

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

ENSEMBLE SHARING - Option Info Initialization

Cached	New
old_TFO_cookie	old_TFO_cookie
old_TFO_failure	old_TFO_failure

7.2. Updates to the new TCB

During the connection, the associated TCB can be updated based on changes to concurrent connections, as shown below:

ENSEMBLE SHARING - Cache Updates

Cached TCB	Current TCB	when?	New Cached TCB
old_MMS_S	curr_MMS_S	OPEN	curr_MMS_S
old_MMS_R	curr_MMS_R	OPEN	curr_MMS_R
old_sendMSS	curr_sendMSS	MSSopt	curr_sendMSS
old_PMTU	curr_PMTU	PMTUD / PLPMTUD	curr_PMTU
old_RTT	curr_RTT	update	rtt_update(old,curr)
old_RTTvar	curr_RTTvar	update	rtt_update(old,curr)
old_ssthresh	curr_ssthresh	update	adjust sum as appropriate
old_sendcwnd	curr_sendcwnd	update	adjust sum as appropriate
old_option	curr_option	(depends)	(option specific)

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

ENSEMBLE SHARING - Option Info Updates

Cached	Current	when?	New Cached
old_TFO_cookie	old_TFO_cookie	ESTAB	old_TFO_cookie
old_TFO_failure	old_TFO_failure	ESTAB	old_TFO_failure

7.3. Discussion

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

Congestion window size and ssthresh aggregation are more complicated in the concurrent case. When there is an ensemble of connections, we need to decide how that ensemble would have shared these variables, in order to derive initial values for new TCBs.

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

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

There are several ways to initialize the congestion window in a new TCB among an ensemble of current connections to a host. Current TCP implementations initialize it to four segments as standard [rfc3390] and 10 segments experimentally [RFC6928]. These approaches assume that new connections should behave as conservatively as possible. The algorithm described in [Ba12] adjusts the initial cwnd depending on the cwnd values of ongoing connections. It is also possible to use sharing mechanisms over long timescales to adapt TCP's initial window automatically, as described further in Appendix C.

8. Compatibility Issues

Here, we discuss various types of problems that may arise with TCB information sharing.

For the congestion and current window information, the initial values computed by TCB interdependence may not be consistent with the long-term aggregate behavior of a set of concurrent connections between the same endpoints. Under conventional TCP congestion control, if a single existing connection has converged to a congestion window of 40 segments, two newly joining concurrent connections assume initial windows of 10 segments [RFC6928], and the current connection's window doesn't decrease to accommodate this additional load and connections can mutually interfere. One example of this is seen on low-bandwidth, high-delay links, where concurrent connections supporting Web traffic can collide because their initial windows were too large, even when set at one segment.

The authors of [Hu12] recommend caching ssthresh for temporal sharing only when flows are long. Some studies suggest that sharing ssthresh between short flows can deteriorate the performance of

individual connections [Hu12, Du16], although this may benefit aggregate network performance.

8.1. Traversing the same network path

TCP is sometimes used in situations where packets of the same host-pair do not always take the same path. Multipath routing that relies on examining transport headers, such as ECMP and LAG [RFC7424], may not result in repeatable path selection when TCP segments are encapsulated, encrypted, or altered - for example, in some Virtual Private Network (VPN) tunnels that rely on proprietary encapsulation. Similarly, such approaches cannot operate deterministically when the TCP header is encrypted, e.g., when using IPsec ESP (although TCB interdependence among the entire set sharing the same endpoint IP addresses should work without problems when the TCP header is encrypted). Measures to increase the probability that connections use the same path could be applied: e.g., the connections could be given the same IPv6 flow label. TCB interdependence can also be extended to sets of host IP address pairs that share the same network path conditions, such as when a group of addresses is on the same LAN (see Section 9).

Traversing the same path is not important for host-specific information such as RWIN and TCP option state, such as TFOinfo. When TCB information is shared across different SYN destination ports, path-related information can be incorrect; however, the impact of this error is potentially diminished if (as discussed here) TCB sharing affects only the transient event of a connection start or if TCB information is shared only within connections to the same SYN destination port. In case of Temporal Sharing, TCB information could also become invalid over time. Because this is similar to the case when a connection becomes idle, mechanisms that address idle TCP connections (e.g., [RFC7661]) could also be applied to TCB cache management, especially when TCP Fast Open is used [RFC7413].

8.2. State dependence

There may be additional considerations to the way in which TCB interdependence rebalances congestion feedback among the current connections, e.g., it may be appropriate to consider the impact of a connection being in Fast Recovery [RFC5681] or some other similar unusual feedback state, e.g., as inhibiting or affecting the calculations described herein.

8.3. Problems with IP sharing

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

9. Implications

There are several implications to incorporating TCB interdependence in TCP implementations. First, it may reduce the need for application-layer multiplexing for performance enhancement [RFC7231]. Protocols like HTTP/2 [RFC7540] avoid connection reestablishment costs by serializing or multiplexing a set of per-host connections across a single TCP connection. This avoids TCP's per-connection OPEN handshake and also avoids recomputing the MSS, RTT, and congestion window values. By avoiding the so-called, "slow-start restart," performance can be optimized [Hu01]. TCB interdependence can provide the "slow-start restart avoidance" of multiplexing, without requiring a multiplexing mechanism at the application layer.

Like the initial version of this document [RFC2140], this update's approach to TCB interdependence focuses on sharing a set of TCBs by updating the TCB state to reduce the impact of transients when connections begin or end. Other mechanisms have since been proposed to continuously share information between all ongoing communication (including connectionless protocols), updating the congestion state during any congestion-related event (e.g., timeout, loss confirmation, etc.) [RFC3124]. By dealing exclusively with transients, TCB interdependence is more likely to exhibit the same behavior as unmodified, independent TCP connections.

9.1. Layering

TCB interdependence pushes some of the TCP implementation from the traditional transport layer (in the ISO model), to the network layer. This acknowledges that some state is in fact per-host-pair or can be per-path as indicated solely by that host-pair. Transport protocols typically manage per-application-pair associations (per

stream), and network protocols manage per-host-pair and path associations (routing). Round-trip time, MSS, and congestion information could be more appropriately handled in a network-layer fashion, aggregated among concurrent connections, and shared across connection instances [RFC3124].

An earlier version of RTT sharing suggested implementing RTT state at the IP layer, rather than at the TCP layer. Our observations describe sharing state among TCP connections, which avoids some of the difficulties in an IP-layer solution. One such problem of an IP layer solution is determining the correspondence between packet exchanges using IP header information alone, where such correspondence is needed to compute RTT. Because TCB sharing computes RTTs inside the TCP layer using TCP header information, it can be implemented more directly and simply than at the IP layer. This is a case where information should be computed at the transport layer but could be shared at the network layer.

9.2. Other possibilities

Per-host-pair associations are not the limit of these techniques. It is possible that TCBs could be similarly shared between hosts on a subnet or within a cluster, because the predominant path can be subnet-subnet, rather than host-host. Additionally, TCB interdependence can be applied to any protocol with congestion state, including SCTP [RFC4960] and DCCP [RFC4340], as well as for individual subflows in Multipath TCP [RFC8684].

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

10. Implementation Observations

The observation that some TCB state is host-pair specific rather than application-pair dependent is not new and is a common engineering decision in layered protocol implementations. Although now deprecated, T/TCP [RFC1644] was the first to propose using caches in order to maintain TCB states (see Appendix A).

The table below describes the current implementation status for some TCB temporal sharing in Linux kernel version 4.6, FreeBSD 10 and Windows as of October 2016. Ensemble sharing is not yet implemented.

KNOWN IMPLEMENTATION STATUS

TCB data	Status
old_MMS_S	Not shared
old_MMS_R	Not shared
old_sendMSS	Cached and shared in Linux (MSS)
old_PMTU	Cached and shared in FreeBSD and Windows (PMTU)
old_RTT	Cached and shared in FreeBSD and Linux
old_RTTvar	Cached and shared in FreeBSD
old_TFOinfo	Cached and shared in Linux and Windows
old_sendcwnd	Not shared
old_ssthresh	Cached and shared in FreeBSD and Linux*

*Note: In FreeBSD, new ssthresh is the mean of curr_ssthresh and previous value if a previous value exists; in Linux, the calculation depends on state and is max(curr_cwnd/2, old_ssthresh) in most cases.

11. Updates to RFC 2140

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

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

New sections have been added to address compatibility issues and implementation observations. The relation of this work to T/TCP has been moved to Appendix A on history, partly to reflect the deprecation of that protocol.

Appendix C has been added to discuss the potential to use temporal sharing over long timescales to adapt TCP's initial window automatically, avoiding the need to periodically revise a single global constant value.

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

12. Security Considerations

These presented implementation methods do not have additional ramifications for explicit attacks. They may be susceptible to denial-of-service attacks if not otherwise secured.

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

Attacks on parameters used only for initialization affect only the transient performance of a TCP connection. For short connections, the performance ramification can approach that of a denial-of-service attack. E.g., if an application changes its TCB to have a false and small window size, subsequent connections will experience performance degradation until their window grew appropriately.

TCB sharing reuses and mixes information from past and current connections. Although reusing information could create a potential for fingerprinting to identify hosts, the mixing reduces that potential. There has been no evidence of fingerprinting based on this technique and it is currently considered safe in that regard.

13. IANA Considerations

There are no IANA implications or requests in this document.

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

14. References

14.1. Normative References

[RFC793] Postel, Jon, "Transmission Control Protocol," Network Working Group RFC-793/STD-7, ISI, Sept. 1981.

[RFC1122] Braden, R. (ed), "Requirements for Internet Hosts -- Communication Layers", RFC-1122, Oct. 1989.

[RFC1191] Mogul, J., Deering, S., "Path MTU Discovery," RFC 1191, Nov. 1990.

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

[RFC4821] Mathis, M., Heffner, J., "Packetization Layer Path MTU Discovery," RFC 4821, Mar. 2007.

[RFC5681] Allman, M., Paxson, V., Blanton, E., "TCP Congestion Control," RFC 5681 (Standards Track), Sep. 2009.

[RFC6298] Paxson, V., Allman, M., Chu, J., Sargent, M., "Computing TCP's Retransmission Timer," RFC 6298, June 2011.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., Jain, A., "TCP Fast Open", RFC 7413, Dec. 2014.

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

[RFC8201] McCann, J., Deering. S., Mogul, J., Hinden, R. (Ed.), "Path MTU Discovery for IP version 6," RFC 8201, Jul. 2017.

14.2. Informative References

[Al10] Allman, M., "Initial Congestion Window Specification", (work in progress), draft-allman-tcpm-bump-initcwnd-00, Nov. 2010.

[Ba12] Barik, R., Welzl, M., Ferlin, S., Alay, O., " LISA: A Linked Slow-Start Algorithm for MPTCP", IEEE ICC, Kuala Lumpur, Malaysia, May 23-27 2016.

[Ba20] Bagnulo, M., Briscoe, B., "ECN++: Adding Explicit Congestion Notification (ECN) to TCP Control Packets", draft-ietf-tcpm-generalized-ecn-06, Oct. 2020.

[Be94] Berners-Lee, T., et al., "The World-Wide Web," Communications of the ACM, V37, Aug. 1994, pp. 76-82.

[Br94] Braden, B., "T/TCP -- Transaction TCP: Source Changes for Sun OS 4.1.3,", Release 1.0, USC/ISI, September 14, 1994.

[Br02] Brownlee, N. and K. Claffy, "Understanding Internet Traffic Streams: Dragonflies and Tortoises", IEEE Communications Magazine p110-117, 2002.

[Co91] Comer, D., Stevens, D., Internetworking with TCP/IP, V2, Prentice-Hall, NJ, 1991.

[Du16] Dukkipati, N., Yuchung C., and Amin V., "Research Impacting the Practice of Congestion Control." ACM SIGCOMM CCR (editorial), on-line post, July 2016.

[FreeBSD] FreeBSD source code, Release 2.10, http://www.freebsd.org/

[Hu01] Hugues, A., Touch, J., Heidemann, J., "Issues in Slow-Start Restart After Idle", draft-hughes-restart-00 (expired), Dec. 2001.

[Hu12] Hurtig, P., Brunstrom, A., "Enhanced metric caching for short TCP flows," 2012 IEEE International Conference on Communications (ICC), Ottawa, ON, 2012, pp. 1209-1213.

[Ja88] Jacobson, V., M. Karels, "Congestion Avoidance and Control", Proc. Sigcomm 1988.

[RFC1644] Braden, R., "T/TCP -- TCP Extensions for Transactions Functional Specification," RFC-1644, July 1994.

[RFC1379] Braden, R., "Transaction TCP -- Concepts," RFC-1379, September 1992.

[RFC2001] Stevens, W., "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms", RFC2001 (Standards Track), Jan. 1997.

[RFC2140] Touch, J., "TCP Control Block Interdependence", RFC 2140, April 1997.

[RFC2414] Allman, M., Floyd, S., Partridge, C., "Increasing TCP's Initial Window", RFC 2414 (Experimental), Sept. 1998.

[RFC2663] Srisuresh, P., Holdrege, M., "IP Network Address Translator (NAT) Terminology and Considerations", RFC-2663, August 1999.

[RFC3390] Allman, M., Floyd, S., Partridge, C., "Increasing TCP's Initial Window," RFC 3390, Oct. 2002.

[RFC3124] Balakrishnan, H., Seshan, S., "The Congestion Manager," RFC 3124, June 2001.

[RFC4340] Kohler, E., Handley, M., Floyd, S., "Datagram Congestion Control Protocol (DCCP)," RFC 4340, Mar. 2006.

[RFC4960] Stewart, R., (Ed.), "Stream Control Transmission Protocol," RFC4960, Sept. 2007.

[RFC5925] Touch, J., Mankin, A., Bonica, R., "The TCP Authentication Option," RFC 5925, June 2010.

[RFC6928] Chu, J., Dukkipati, N., Cheng, Y., Mathis, M., "Increasing TCP's Initial Window," RFC 6928, Apr. 2013.

[RFC7231] Fielding, R., J. Reshke, Eds., "HTTP/1.1 Semantics and Content," RFC-7231, June 2014.

[RFC7323] Borman, D., B. Braden, V. Jacobson, R. Scheffenegger (Ed.), "TCP Extensions for High Performance," RFC 7323, Sept. 2014.

[RFC7424] Krishnan, R., Yong, L., Ghanwani, A., So, N., Khasnabish, B., "Mechanisms for Optimizing Link Aggregation Group (LAG) and Equal-Cost Multipath (ECMP) Component Link Utilization in Networks", RFC 7424, Jan. 2015

[RFC7540] Belshe, M., Peon, R., Thomson, M., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, May 2015.

[RFC7661] Fairhurst, G., Sathiaseelan, A., Secchi, R., "Updating TCP to Support Rate-Limited Traffic", RFC 7661, Oct. 2015.

[RFC8684] Ford, A., Raiciu, C., Handley, M., Bonaventure, O., Paasch, C., "TCP Extensions for Multipath Operation with Multiple Addresses," RFC 8684, Mar. 2020.

15. Acknowledgments

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

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

16. Change log

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

ietf-07:

- Update per id-nits and normative language for consistency

ietf-06:

- Address WGLC comments

ietf-05:

- Correction of typographic errors, expansion of terminology

ietf-04:

- Fix internal cross-reference errors that appeared in ietf-02
- Updated tables to re-center; clarified text

ietf-03:

- Correction of typographic errors, minor rewording in appendices

ietf-02:

- Minor reorganization and correction of typographic errors
- Added text to address fingerprinting in Security section
- Now retains Appendix B and body option tables upon publication

ietf-01:

- Added Appendix C to address long-timescale temporal adaptation

ietf-00:

- Re-issued as draft-ietf-tcpm-2140bis due to WG adoption.
- Cleaned orphan references to T/TCP, removed incomplete refs
- Moved references to informative section and updated Sec 2
- Updated to clarify no impact to interoperability
- Updated appendix B to avoid 2119 language

06:

- Changed to update 2140, cite it normatively, and summarize the updates in a separate section

05:

- Fixed some TBDs.

04:

- Removed BCP-style recommendations and fixed some TBDs.

03:

- Updated Touch's affiliation and address information

02:

- Stated that our OS implementation overview table only covers temporal sharing.

- Correctly reflected sharing of old_RTT in Linux in the implementation overview table.

- Marked entries that are considered safe to share with an asterisk (suggestion was to split the table)

- Discussed correct host identification: NATs may make IP addresses the wrong input, could e.g., use HTTP cookie.

- Included MMS_S and MMS_R from RFC1122; fixed the use of MSS and MTU

- Added information about option sharing, listed options in Appendix B

Authors' Addresses

Joe Touch
Manhattan Beach, CA 90266
USA

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

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

Phone: +47 22 85 24 20
Email: michawe@ifi.uio.no

Safiqul Islam
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Phone: +47 22 84 08 37
Email: safiquli@ifi.uio.no

Appendix A: TCB Sharing History

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

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

Temporal sharing of cached TCB data was originally implemented in the SunOS 4.1.3 T/TCP extensions [Br94] and the FreeBSD port of same [FreeBSD]. As mentioned before, only the MSS and RTT parameters were cached, as originally specified in [RFC1379]. Later discussion of T/TCP suggested including congestion control parameters in this cache; for example, [RFC1644] (Section 3.1) hints at initializing the congestion window to the old window size.

Appendix B: TCP Option Sharing and Caching

In addition to the options that can be cached and shared, this memo also lists known options for which state is unsafe to be kept. This list is not intended to be authoritative or exhaustive.

Obsolete (unsafe to keep state):

ECHO

ECHO REPLY

PO Conn permitted

PO service profile

CC

CC.NEW

CC.ECHO

Alt CS req

Alt CS data

No state to keep:

EOL

NOP

WS

SACK

TS

MD5

TCP-AO

EXP1

EXP2

Unsafe to keep state:

- Skeeter (DH exchange, known to be vulnerable)
- Bubba (DH exchange, known to be vulnerable)
- Trailer CS
- SCPS capabilities
- S-NACK
- Records boundaries
- Corruption experienced
- SNAP
- TCP Compression
- Quickstart response
- UTO
- MPTCP negotiation success (see below for negotiation failure)
- TFO negotiation success (see below for negotiation failure)

Safe but optional to keep state:

- MPTCP negotiation failure (to avoid negotiation retries)
- MSS
- TFO negotiation failure (to avoid negotiation retries)

Safe and necessary to keep state:

- TFP cookie (if TFO succeeded in the past)

Appendix C: Automating the Initial Window in TCP over Long Timescales

C.1. Introduction

Temporal sharing, as described earlier in this document, builds on the assumption that multiple consecutive connections between the same host pair are somewhat likely to be exposed to similar environment characteristics. The stored information can therefore become invalid over time, and suitable precautions should be taken (this is discussed further in section 8.1). However, there are also cases where it can make sense to use much longer-term measurements of TCP connections to gradually influence TCP parameters. This appendix describes an example of such a case.

TCP's congestion control algorithm uses an initial window value (IW), both as a starting point for new connections and as an upper limit for restarting after an idle period [RFC5681][RFC7661]. This value has evolved over time, originally one maximum segment size (MSS), and increased to the lesser of four MSS or 4,380 bytes [RFC3390][RFC5681]. For a typical Internet connection with a maximum transmission unit (MTU) of 1500 bytes, this permits three segments of 1,460 bytes each.

The IW value was originally implied in the original TCP congestion control description and documented as a standard in 1997 [RFC2001][Ja88]. The value was updated in 1998 experimentally and moved to the standards track in 2002 [RFC2414][RFC3390]. In 2013, it was experimentally increased to 10 [RFC6928].

This appendix discusses how TCP can objectively measure when an IW is too large, and that such feedback should be used over long timescales to adjust the IW automatically. The result should be safer to deploy and might avoid the need to repeatedly revisit IW over time.

Note that this mechanism attempts to make the IW more adaptive over time. It can increase the IW beyond that which is currently recommended for widescale deployment, and so its use should be carefully monitored.

C.2. Design Considerations

TCP's IW value has existed statically for over two decades, so any solution to adjusting the IW dynamically should have similarly stable, non-invasive effects on the performance and complexity of TCP. In order to be fair, the IW should be similar for most machines on the public Internet. Finally, a desirable goal is to develop a

self-correcting algorithm, so that IW values that cause network problems can be avoided. To that end, we propose the following design goals:

- Impart little to no impact to TCP in the absence of loss, i.e., it should not increase the complexity of default packet processing in the normal case.
- Adapt to network feedback over long timescales, avoiding values that persistently cause network problems.
- Decrease the IW in the presence of sustained loss of IW segments, as determined over a number of different connections.
- Increase the IW in the absence of sustained loss of IW segments, as determined over a number of different connections.
- Operate conservatively, i.e., tend towards leaving the IW the same in the absence of sufficient information, and give greater consideration to IW segment loss than IW segment success.

We expect that, without other context, a good IW algorithm will converge to a single value, but this is not required. An endpoint with additional context or information, or deployed in a constrained environment, can always use a different value. In specific, information from previous connections, or sets of connections with a similar path, can already be used as context for such decisions (as noted in the core of this document).

However, if a given IW value persistently causes packet loss during the initial burst of packets, it is clearly inappropriate and could be inducing unnecessary loss in other competing connections. This might happen for sites behind very slow boxes with small buffers, which may or may not be the first hop.

C.3. Proposed IW Algorithm

Below is a simple description of the proposed IW algorithm. It relies on the following parameters:

- MinIW = 3 MSS or 4,380 bytes (as per [RFC3390])
- MaxIW = 10 MSS (as per [RFC6928])
- MulDecr = 0.5
- AddIncr = 2 MSS

- Threshold = 0.05

We assume that the minimum IW (MinIW) should be as currently specified [RFC3390]. The maximum IW can be set to a fixed value (as recommended in [RFC6928]) or set based on a schedule if trusted time references are available [Al10]; here we prefer a fixed value. We also propose to use an AIMD algorithm, with increase and decreases as noted.

Although these parameters are somewhat arbitrary, their initial values are not important except that the algorithm is AIMD and the MaxIW should not exceed that recommended for other systems on the Internet. Current proposals, including default current operation, are degenerate cases of the algorithm below for given parameters - notably MulDec = 1.0 and AddIncr = 0 MSS, thus disabling the automatic part of the algorithm.

The proposed algorithm is as follows:

1. On boot:

   ```
   IW = MaxIW; # assume this is in bytes, and an even number of MSS
   ```

2. Upon starting a new connection:

   ```
   CWND = IW;
   conncount++;
   IWnotchecked = 1; # true
   ```

3. During a connection's SYN-ACK processing, if SYN-ACK includes ECN (as similarly addressed in Sec 5 of ECN++ for TCP [Ba20]), treat as if the IW is too large:

   ```
   if (IWnotchecked && (synackecn == 1)) {
      losscount++;
      IWnotchecked = 0; # never check again
   }
   ```

4. During a connection, if retransmission occurs, check the seqno of the outgoing packet (in bytes) to see if the resent segment fixes an IW loss:

```
   if (Retransmitting && IWnotchecked && ((ISN - seqno) < IW))) {
      losscount++;
      IWnotchecked = 0; # never do this entire "if" again
   } else {
      IWnotchecked = 0; # you're beyond the IW so stop checking
   }
```

5. Once every 1000 connections, as a separate process (i.e., not as part of processing a given connection):

```
   if (conncount > 1000) {
      if (losscount/conncount > threshold) {
         # the number of connections with errors is too high
         IW = IW * MulDecr;
      } else {
         IW = IW + AddIncr;
      }
   }
```

As presented, this algorithm can yield a false positive when the sequence number wraps around, e.g., the code might increment losscount in step 4 when no loss occurred or fail to increment losscount when a loss did occur. This can be avoided using either PAWS [RFC7323] context or internal extended sequence number representations (as in TCP-AO [RFC5925]). Alternately, false positives can be tolerated because they are expected to be infrequent and thus will not significantly impact the algorithm.

A number of additional constraints need to be imposed if this mechanism is implemented to ensure that it defaults values that comply with current Internet standards, is conservative in how it extends those values, and returns to those values in the absence of positive feedback (i.e., success). To that end, we recommend the following list of example constraints:

>> The automatic IW algorithm MUST initialize MaxIW a value no larger than the currently recommended Internet default, in the absence of other context information.

Thus, if there are too few connections to make a decision or if there is otherwise insufficient information to increase the IW, then the MaxIW defaults to the current recommended value.

>> An implementation MAY allow the MaxIW to grow beyond the currently recommended Internet default, but not more than 2 segments per calendar year.

Thus, if an endpoint has a persistent history of successfully transmitting IW segments without loss, then it is allowed to probe the Internet to determine if larger IW values have similar success. This probing is limited and requires a trusted time source, otherwise the MaxIW remains constant.

>> An implementation MUST adjust the IW based on loss statistics at least once every 1000 connections.

An endpoint needs to be sufficiently reactive to IW loss.

>> An implementation MUST decrease the IW by at least one MSS when indicated during an evaluation interval.

An endpoint that detects loss needs to decrease its IW by at least one MSS, otherwise it is not participating in an automatic reactive algorithm.

>> An implementation MUST increase by no more than 2 MSS per evaluation interval.

An endpoint that does not experience IW loss needs to probe the network incrementally.

>> An implementation SHOULD use an IW that is an integer multiple of 2 MSS.

The IW should remain a multiple of 2 MSS segments, to enable efficient ACK compression without incurring unnecessary timeouts.

>> An implementation MUST decrease the IW if more than 95% of connections have IW losses.

Again, this is to ensure an implementation is sufficiently reactive.

>> An implementation MAY group IW values and statistics within subsets of connections. Such grouping MAY use any information about connections to form groups except loss statistics.

There are some TCP connections which might not be counted at all, such as those to/from loopback addresses, or those within the same subnet as that of a local interface (for which congestion control is sometimes disabled anyway). This may also include connections that terminate before the IW is full, i.e., as a separate check at the time of the connection closing.

The period over which the IW is updated is intended to be a long timescale, e.g., a month or so, or 1,000 connections, whichever is longer. An implementation might check the IW once a month, and simply not update the IW or clear the connection counts in months where the number of connections is too small.

C.4. Discussion

There are numerous parameters to the above algorithm that are compliant with the given requirements; this is intended to allow variation in configuration and implementation while ensuring that all such algorithms are reactive and safe.

This algorithm continues to assume segments because that is the basis of most TCP implementations. It might be useful to consider revising the specifications to allow byte-based congestion given sufficient experience.

The algorithm checks for IW losses only during the first IW after a connection start; it does not check for IW losses elsewhere the IW is used, e.g., during slow-start restarts.

>> An implementation MAY detect IW losses during slow-start restarts in addition to losses during the first IW of a connection. In this case, the implementation MUST count each restart as a "connection" for the purposes of connection counts and periodic rechecking of the IW value.

False positives can occur during some kinds of segment reordering, e.g., that might trigger spurious retransmissions even without a true segment loss. These are not expected to be sufficiently common to dominate the algorithm and its conclusions.

This mechanism does require additional per-connection state, which is currently common in some implementations, and is useful for other reasons (e.g., the ISN is used in TCP-AO [RFC5925]). The mechanism also benefits from persistent state kept across reboots, as would be other state sharing mechanisms (e.g., TCP Control Block Sharing [RFC2140]). The mechanism is inspired by RFC 2140's use of information across connections.

The receive window (RWIN) is not involved in this calculation. The size of RWIN is determined by receiver resources and provides space to accommodate segment reordering. It is not involved with congestion control, which is the focus of this document and its management of the IW.

C.5. Observations

The IW may not converge to a single, global value. It also may not converge at all, but rather may oscillate by a few MSS as it repeatedly probes the Internet for larger IWs and fails. Both properties are consistent with TCP behavior during each individual connection.

This mechanism assumes that losses during the IW are due to IW size. Persistent errors that drop packets for other reasons – e.g., OS bugs, can cause false positives. Again, this is consistent with TCP's basic assumption that loss is caused by congestion and requires backoff. This algorithm treats the IW of new connections as a long-timescale backoff system.

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: July 10, 2021

P. Balasubramanian
Y. Huang
M. Olson
Microsoft

January 6, 2021

HyStart++: Modified Slow Start for TCP
draft-ietf-tcpm-hystartplusplus-01

Abstract

This doument describes HyStart++, a simple modification to the slow start phase of TCP congestion control algorithms. Traditional slow start can cause overshotting of the ideal send rate and cause large packet loss within a round-trip time which results in poor performance. HyStart++ combines the use of one variant of HyStart and Limited Slow Start (LSS) to prevent overshooting of the ideal sending rate, while also mitigating poor performance which can result from false positives when HyStart is used alone.

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 July 10, 2021.

Copyright Notice

Table of Contents

1. Introduction

[RFC5681] describes the slow start congestion control algorithm for TCP. The slow start algorithm is used when the congestion window (cwnd) is less than the slow start threshold (ssthresh). During slow start, in absence of packet loss signals, TCP sender increases cwnd exponentially to probe the network capacity. Such a fast growth can lead to overshooting the ideal sending rate and cause significant packet loss. This is counter-productive for the TCP flow itself, and also impacts the rest of the traffic sharing the bottleneck link. TCP has several mechanisms for loss recovery, but they are only effective for moderate loss. When these techniques are unable to recover lost packets, a last-resort retransmission timeout (RTO) is used to trigger packet recovery. In most operating systems, the minimum RTO is set to a large value (200 msec or 300 msec) to prevent spurious timeouts. This results in a long idle time which drastically impairs flow completion times.

HyStart++ adds delay increase as a signal to exit slow start before any packet loss occurs. This is one of two algorithms specified in [HyStart]. After the HyStart delay algorithm finds an exit point, LSS is used in conjunction with congestion avoidance for further congestion window increases until the first packet loss is detected. HyStart++ reduces packet loss and retransmissions, and improves goodput in lab measurements as well as real world deployments.

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 [RFC2119].

3. Definitions

We repeat here some definition from [RFC5681] to aid the reader.

SENDER MAXIMUM SEGMENT SIZE (SMSS): The SMSS is the size of the largest segment that the sender can transmit. This value can be based on the maximum transmission unit of the network, the path MTU discovery [RFC1191, RFC4821] algorithm, RMSS (see next item), or other factors. The size does not include the TCP/IP headers and options.

RECEIVER MAXIMUM SEGMENT SIZE (RMSS): The RMSS is the size of the largest segment the receiver is willing to accept. This is the value specified in the MSS option sent by the receiver during connection startup. Or, if the MSS option is not used, it is 536 bytes [RFC1122]. The size does not include the TCP/IP headers and options.

RECEIVER WINDOW (rwnd): The most recently advertised receiver window.

CONGESTION WINDOW (cwnd): A TCP state variable that limits the amount of data a TCP can send. At any given time, a TCP MUST NOT send data with a sequence number higher than the sum of the highest acknowledged sequence number and the minimum of cwnd and rwnd.

4. HyStart++ Algorithm

4.1. Use of HyStart Delay Increase and Limited Slow Start

[HyStart] specifies two algorithms (a "Delay Increase" algorithm and an "Inter-Packet Arrival" algorithm) to be run in parallel to detect that the sending rate has reached capacity. In practice, the Inter-Packet Arrival algorithm does not perform well and is not able to detect congestion early, primarily due to ACK compression. The idea of the Delay Increase algorithm is to look for RTT spikes, which suggest that the bottleneck buffer is filling up.

After the HyStart "Delay Increase" algorithm triggers an exit from slow start, LSS (described in [RFC3742]) is used to increase Cwnd until congestion is observed. LSS is used because the HyStart exit is often premature as a result of RTT fluctuations or transient queue buildup. LSS grows the cwnd fast but much slower than traditional

slow start. LSS helps avoid massive packet losses and subsequent time spent in loss recovery or retransmission timeout.

4.2. Algorithm Details

We assume that Appropriate Byte Counting (as described in [RFC3465]) is in use and L is the cwnd increase limit. The choice of value of L is up to the implementation.

A round is chosen to be approximately the Round-Trip Time (RTT). Round can be approximated using sequence numbers as follows:

Define windowEnd as a sequence number initialize to SND.UNA

When windowEnd is ACKed, the current round ends and windowEnd is set to SND.NXT

At the start of each round during slow start:

```
lastRoundMinRTT = currentRoundMinRTT

currentRoundMinRTT = infinity

rttSampleCount = 0
```

For each arriving ACK in slow start, where N is the number of previously unacknowledged bytes acknowledged in the arriving ACK and w:

Update the cwnd

```
cwnd = cwnd + min (N, L * SMSS)
```

Keep track of minimum observed RTT

```
currentRoundMinRTT = min(currentRoundMinRTT, currRTT)
```

where currRTT is the measured RTT based on the incoming ACK

```
rttSampleCount += 1
```

For rounds where cwnd is at or higher than LOW_CWND and N_RTT_SAMPLE RTT samples have been obtained, check if delay increase triggers slow start exit

```
if (cwnd >= (LOW_CWND * SMSS) AND rttSampleCount >=
N_RTT_SAMPLE)
```

```
RttThresh = clamp(MIN_RTT_THRESH, lastRoundMinRTT / 8,
MAX_RTT_THRESH)

if (currentRoundMinRTT >= (lastRoundMinRTT + RttThresh))

   ssthresh = cwnd

   exit slow start and enter LSS
```

For each arriving ACK in LSS, where N is the number of previously unacknowledged bytes acknowledged in the arriving ACK:

```
K = cwnd / (LSS_DIVISOR * ssthresh)

cwnd = max(cwnd + (min (N, L * SMSS) / K), CA_cwnd())
```

CA_cwnd() denotes the cwnd that a congestion control algorithm would have increased to if congestion avoidance started instead of LSS. LSS grows cwnd very fast but for long-lived flows in high BDP networks, the congestion avoidance algorithm could increase cwnd much faster. For example, CUBIC congestion avoidance [RFC8312] in convex region can ramp up cwnd rapidly. Taking the max can help improve performance when exiting slow start prematurely.

HyStart++ ends when congestion is observed.

4.3. Tuning constants

It is RECOMMENDED that a HyStart++ implementation use the following constants:

```
LOW_CWND = 16

MIN_RTT_THRESH = 4 msec

MAX_RTT_THRESH = 16 msec

LSS_DIVISOR = 0.25

N_RTT_SAMPLE = 8
```

These constants have been determined with lab measurements and real world deployments. An implementation MAY tune them for different network characteristics.

Using smaller values of LOW_CWND will cause the algorithm to kick in before the last round RTT can be measured, particularly if the implementation uses an initial cwnd of 10 MSS. Higher values will

delay the detection of delay increase and reduce the ability of HyStart++ to prevent overshoot problems.

The delay increase sensitivity is determined by MIN_RTT_THRESH and MAX_RTT_THRESH. Smaller values of MIN_RTT_THRESH may cause spurious exits from slow start. Larger values of MAX_RTT_THRESH may result in slow start not exiting until loss is encountered for connections on large RTT paths.

A TCP implementation is required to take at least one RTT sample each round. Using lower values of N_RTT_SAMPLE will lower the accuracy of the measured RTT for the round; higher values will improve accuracy at the cost of more processing.

The maximum value of LSS_DIVISOR SHOULD NOT exceed 0.5, which is the value recommended in [RFC3742]. Otherwise the cwnd growth could again become too aggressive and cause ideal send rate overshoot. Smaller values will cause the algorithm to be less aggressive and may leave some cwnd growth on the table.

An implementation SHOULD use HyStart++ only for the initial slow start and fall back to using traditional slow start for the remainder of the connection lifetime. This is acceptable because subsequent slow starts will use the discovered ssthresh value to exit slow start. An implementation MAY use HyStart++ to grow the restart window ([RFC5681]) after a long idle period.

5. Deployments and Performance Evaluations

As of the time of writing, HyStart++ has been default enabled for all TCP connections in Windows for two years. The original Hystart has been default-enabled for all TCP connections in Linux TCP for a decade.

In lab measurements with Windows TCP, HyStart++ shows both goodput improvements as well as reductions in packet loss and retransmissions. For example across a variety of tests on a 100 Mbps link with a bottleneck buffer size of bandwidth-delay product, HyStart++ reduces bytes retransmitted by 50% and retransmission timeouts by 36%.

In an A/B test across a large Windows device population, out of 52 billion TCP connections, 0.7% of connections move from 1 RTO to 0 RTOs and another 0.7% connections move from 2 RTOs to 1 RTO with HyStart++. This test did not focus on send heavy connections and the impact on send heavy connections is likely much higher. We plan to conduct more such production experiments to gather more data in the future.

6. Security Considerations

HyStart++ enhances slow start and inherits the general security considerations discussed in [RFC5681].

7. IANA Considerations

This document has no actions for IANA.

8. Acknowledgements

Neal Cardwell suggested the idea of using the maximum of cwnd value computed by LSS and congestion avoidance after exiting slow start.

9. References

9.1. Normative References

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

[RFC3465] Allman, M., "TCP Congestion Control with Appropriate Byte Counting (ABC)", RFC 3465, DOI 10.17487/RFC3465, February 2003, <https://www.rfc-editor.org/info/rfc3465>.

[RFC3742] Floyd, S., "Limited Slow-Start for TCP with Large Congestion Windows", RFC 3742, DOI 10.17487/RFC3742, March 2004, <https://www.rfc-editor.org/info/rfc3742>.

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

9.2. Informative References

[HyStart] Ha, S. and I. Ree, "Hybrid Slow Start for High-Bandwidth and Long-Distance Networks", DOI 10.1145/1851182.1851192, International Workshop on Protocols for Fast Long-Distance Networks, 2008, <https://pdfs.semanticscholar.org/25e9/ef3f03315782c7f1cbcd31b587857adae7d1.pdf>.

[RFC8312] Rhee, I., Xu, L., Ha, S., Zimmermann, A., Eggert, L., and R. Scheffenegger, "CUBIC for Fast Long-Distance Networks", RFC 8312, DOI 10.17487/RFC8312, February 2018, <https://www.rfc-editor.org/info/rfc8312>.

Authors' Addresses

Praveen Balasubramanian
Microsoft
One Microsoft Way
Redmond, WA 98052
USA

Phone: +1 425 538 2782
Email: pravb@microsoft.com

Yi Huang
Microsoft

Phone: +1 425 703 0447
Email: huanyi@microsoft.com

Matt Olson
Microsoft

Phone: +1 425 538 8598
Email: maolson@microsoft.com

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

Y. Cheng
N. Cardwell
N. Dukkipati
P. Jha
Google, Inc

December 22, 2020

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

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on June 25, 2021.

Table of Contents

1. Terminology

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

2. Introduction

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

2.1. Background

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

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

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

2.2. Motivation

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

1. Packet drops for short flows or at the end of an application data flight. When the sender is limited by the application (e.g. structured request/response traffic), segments lost at the end of the application data transfer often can only be recovered by RTO. Consider an example of losing only the last segment in a flight of 100 segments. Lacking any DUPACK, the sender RTO expires and reduces the congestion window to 1, and raises the congestion window to just 2 after the loss repair is acknowledged. In contrast, any single segment loss occurring between the first and the 97th segment would result in fast recovery, which would only cut the window in half.

2. Lost retransmissions. Heavy congestion or traffic policers can cause retransmissions to be lost. Lost retransmissions cause a resort to RTO recovery, since DUPACK-counting does not detect the loss of the retransmissions. Then the slow start after RTO recovery could cause burst losses again that severely degrades performance [POLICER16].

3. Packet reordering. In this document, "reordering" refers to the events where segments are delivered at the TCP receiver in a chronological order different from their chronological transmission order. Link-layer protocols (e.g., 802.11 block ACK), link bonding, or routers' internal load-balancing (e.g., ECMP) can deliver TCP segments out of order. The degree of such reordering is usually within the order of the path round trip time. If the reordering degree is beyond DupThresh, DUPACK-counting can cause a spurious fast recovery and unnecessary

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

3. RACK-TLP high-level design

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

3.1. RACK: time-based loss inferences from ACKs

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

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

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

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

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

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

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

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

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

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

3.3.1. Reordering design rationale

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

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

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

1. Application-limited flights where the last non-full-sized segment is delivered first and then the remaining full-sized segments in the flight are delivered in order. This reordering pattern can occur when segments traverse parallel forwarding paths. In such scenarios the degree of reordering in packet distance is one segment less than the flight size.

2. A flight of segments that are delivered partially out of order. One cause for this pattern is wireless link-layer retransmissions with an inadequate reordering buffer at the receiver. In such scenarios, the wireless sender sends the data packets in order initially, but some are lost and then recovered by link-layer retransmissions; the wireless receiver delivers the TCP data packets in the order they are received, due to the inadequate reordering buffer. The random wireless transmission errors in such scenarios cause the reordering degree, expressed in packet distance, to have highly variable values up to the flight size.

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

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

1. Another data segment sent later than S has been delivered

2. S has not been delivered after the estimated round-trip time plus the reordering window

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

3.3.2. Reordering window adaptation

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

1. The reordering window SHOULD be set to zero if no reordering has been observed on the connection so far, and either (a) three segments have been delivered out of order since the last recovery or (b) the sender is already in fast or RTO recovery. Otherwise, the reordering window SHOULD start from a small fraction of the round trip time, or zero if no round trip time estimate is available.

2. The RACK reordering window SHOULD adaptively increase (using the algorithm in "Step 4: Update RACK reordering window", below) if the sender receives a Duplicate Selective Acknowledgement (DSACK) option [RFC2883]. Receiving a DSACK suggests the sender made a spurious retransmission, which may have been due to the reordering window being too small.

3. The RACK reordering window MUST be bounded and this bound SHOULD be SRTT.

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

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

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

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

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

```
Event  TCP DATA SENDER                          TCP DATA RECEIVER
_____  ___________________________________________________________

  1.   Send P0, P1, P2, P3          -->
       [P1, P2, P3 dropped by network]

  2.                                <--          Receive P0, ACK P0

  3a.  2RTTs after (2), TLP timer fires
  3b.  TLP: retransmits P3          -->

  4.                                <--         Receive P3, SACK P3

  5a.  Receive SACK for P3
  5b.  RACK: marks P1, P2 lost
  5c.  Retransmit P1, P2            -->
       [P1 retransmission dropped by network]

  6.                                <--    Receive P2, SACK P2 & P3

  7a.  RACK: marks P1 retransmission lost
  7b.  Retransmit P1                -->

  8.                                <--          Receive P1, ACK P3
```

Figure 1. RACK-TLP protocol example

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

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

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

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

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

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

3.6. Design Summary

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

4. Requirements

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

1. The connection MUST use selective acknowledgment (SACK) options [RFC2018], and the sender MUST keep SACK scoreboard information on a per-connection basis ("SACK scoreboard" has the same meaning here as in [RFC6675] section 3).

2. For each data segment sent, the sender MUST store its most recent transmission time with a timestamp whose granularity is finer than 1/4 of the minimum RTT of the connection. At the time of writing, microsecond resolution is suitable for intra-datacenter traffic and millisecond granularity or finer is suitable for the Internet. Note that RACK-TLP can be implemented with TSO (TCP Segmentation Offload) support by having multiple segments in a TSO aggregate share the same timestamp.

3. RACK DSACK-based reordering window adaptation is RECOMMENDED but is not required.

4. TLP requires RACK.

5. Definitions

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

5.1. Terms

These terms are used to explain variables and algorithms below:

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

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

5.2. Per-segment variables

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

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

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

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

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

5.3. Per-connection variables

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.4. Per-connection timers

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

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

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

6. RACK Algorithm Details

6.1. Upon transmitting a data segment

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

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

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

6.2. Upon receiving an ACK

Step 1: Update RACK.min_RTT.

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

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

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

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

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

2. The segment was last retransmitted less than RACK.min_rtt ago.

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

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

Step 2 may be summarized in pseudocode as:

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

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

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

Step 3: Detect data segment reordering

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

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

Step 4: Update RACK reordering window

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

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

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

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

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

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

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

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

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

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

```
RACK_update_reo_wnd():

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

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

Step 5: Detect losses.

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

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

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

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

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

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

Since RACK.RTT >= (RACK.ack_ts - RACK.xmit_ts), the previous equation reduces to saying that the sender can declare a segment lost when:

now >= Segment.xmit_ts + RACK.reo_wnd + RACK.rtt

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

Segment.xmit_ts + RACK.rtt + RACK.reo_wnd - now <= 0

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

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

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

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

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

6.3. Upon RTO expiration

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

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

7. TLP Algorithm Details

7.1. Initializing state

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

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

7.2. Scheduling a loss probe

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

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

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

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

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

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

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

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

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

7.3. Sending a loss probe upon PTO expiration

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

1. First, there is no other previous loss probe still in flight. This ensures that at any given time the sender has at most one additional packet in flight beyond the congestion window limit. This invariant is maintained using the state variable TLP.end_seq, which indicates the latest unacknowledged TLP loss probe's ending sequence. It is reset when the loss probe has been acknowledged or is deemed lost or irrelevant.

2. Second, the sender has obtained an RTT measurement since the last loss probe was transmitted, or, if the sender has not yet sent a loss probe on this connection, since the start of the connection. This condition ensures that loss probe retransmissions do not prevent taking the RTT samples necessary to adapt SRTT to an increase in path RTT.

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

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

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

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

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

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

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

```
TLP_send_probe():

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

7.4. Detecting losses using the ACK of the loss probe

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

7.4.1. General case: detecting packet losses using RACK

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

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

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

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

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

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

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

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

8. Managing RACK-TLP timers

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

9. Discussion

9.1. Advantages and disadvantages

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

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

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

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

3. Packet reordering: Consider a simple reordering event where a flight of segments are sent as (P1, P2, P3). P1 and P2 carry a full payload of MSS octets, but P3 has only a 1-octet payload. Suppose the sender has detected reordering previously and thus RACK.reo_wnd is min_RTT/4. Now P3 is reordered and delivered first, before P1 and P2. As long as P1 and P2 are delivered within min_RTT/4, RACK will not consider P1 and P2 lost. But if P1 and P2 are delivered outside the reordering window, then RACK will still spuriously mark P1 and P2 lost.

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

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

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

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

9.2. Relationships with other loss recovery algorithms

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

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

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

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

9.3. Interaction with congestion control

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

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

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

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

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

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

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

9.4. TLP recovery detection with delayed ACKs

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

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

9.5. RACK-TLP for other transport protocols

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

10. Security Considerations

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

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

11. IANA Considerations

This document makes no request of IANA.

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

12. Acknowledgments

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

13. References

13.1. Normative References

[RFC2018] Mathis, M. and J. Mahdavi, "TCP Selective Acknowledgment Options", RFC 2018, October 1996.

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

[RFC2883] Floyd, S., Mahdavi, J., Mathis, M., and M. Podolsky, "An Extension to the Selective Acknowledgement (SACK) Option for TCP", RFC 2883, July 2000.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, September 2009.

[RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, June 2011.

[RFC6675] Blanton, E., Allman, M., Wang, L., Jarvinen, I., Kojo, M., and Y. Nishida, "A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP", RFC 6675, August 2012.

[RFC7323] Borman, D., Braden, B., Jacobson, V., and R. Scheffenegger, "TCP Extensions for High Performance", September 2014.

[RFC793] Postel, J., "Transmission Control Protocol", September 1981.

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

13.2. Informative References

[DMCG11] Dukkipati, N., Matthis, M., Cheng, Y., and M. Ghobadi, "Proportional Rate Reduction for TCP", ACM SIGCOMM Conference on Internet Measurement , 2011.

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

[POLICER16] Flach, T., Papageorge, P., Terzis, A., Pedrosa, L., Cheng, Y., Karim, T., Katz-Bassett, E., and R. Govindan, "An Analysis of Traffic Policing in the Web", ACM SIGCOMM , 2016.

[QUIC-LR] Iyengar, J. and I. Swett, "QUIC Loss Detection and Congestion Control", draft-ietf-quic-recovery (work in progress), Octobor 2020.

[RFC3042] Allman, M., Balakrishnan, H., and S. Floyd, "Enhancing TCP's Loss Recovery Using Limited Transmit", January 2001.

[RFC3522] Ludwig, R. and M. Meyer, "The Eifel Detection Algorithm for TCP", April 2003.

[RFC4653] Bhandarkar, S., Reddy, A., Allman, M., and E. Blanton, "Improving the Robustness of TCP to Non-Congestion Events", August 2006.

[RFC5682] Sarolahti, P., Kojo, M., Yamamoto, K., and M. Hata, "Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP", RFC 5682, September 2009.

[RFC5827] Allman, M., Ayesta, U., Wang, L., Blanton, J., and P. Hurtig, "Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)", RFC 5827, April 2010.

[RFC6937] Mathis, M., Dukkipati, N., and Y. Cheng, "Proportional Rate Reduction for TCP", May 2013.

[RFC7765] Hurtig, P., Brunstrom, A., Petlund, A., and M. Welzl, "TCP and SCTP RTO Restart", February 2016.

[SCWA99] Savage, S., Cardwell, N., Wetherall, D., and T. Anderson, "TCP Congestion Control With a Misbehaving Receiver", ACM Computer Communication Review, 29(5) , 1999.

[Sprout] Winstein, K., Sivaraman, A., and H. Balakrishnan, "Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks", USENIX Symposium on Networked Systems Design and Implementation (NSDI) , 2013.

Authors' Addresses

Yuchung Cheng
Google, Inc

Email: ycheng@google.com

Neal Cardwell
Google, Inc

Email: ncardwell@google.com

Nandita Dukkipati
Google, Inc

Email: nanditad@google.com

Priyaranjan Jha
Google, Inc

Email: priyarjha@google.com

Internet Engineering Task Force
Internet-Draft
Obsoletes: 793, 879, 2873, 6093, 6429,
6528, 6691 (if approved)
Updates: 5961, 1122 (if approved)
Intended status: Standards Track
Expires: July 25, 2021

W. Eddy, Ed.
MTI Systems

January 21, 2021

Transmission Control Protocol (TCP) Specification
draft-ietf-tcpm-rfc793bis-20

Abstract

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

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

Status of This Memo

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

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

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

This Internet-Draft will expire on July 25, 2021.

Table of Contents

1. Purpose and Scope

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

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

For several decades, RFC 793 plus a number of other documents have combined to serve as the core specification for TCP [48]. Over time, a number of errata have been filed against RFC 793, as well as deficiencies in security, performance, and many other aspects. The number of enhancements has grown over time across many separate documents. These were never accumulated together into a comprehensive update to the base specification.

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

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

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

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

2. Introduction

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

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

The "TCP Roadmap" [48] provides a more extensive guide to the RFCs that define TCP and describe various important algorithms. The TCP Roadmap contains sections on strongly encouraged enhancements that improve performance and other aspects of TCP beyond the basic

operation specified in this document. As one example, implementing congestion control (e.g. [34]) is a TCP requirement, but is a complex topic on its own, and not described in detail in this document, as there are many options and possibilities that do not impact basic interoperability. Similarly, most TCP implementations today include the high-performance extensions in [46], but these are not strictly required or discussed in this document. Multipath considerations for TCP are also specified separately in [54].

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

2.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [4][13] when, and only when, they appear in all capitals, as shown here.

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

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

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

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

2.2. Key TCP Concepts

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

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

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

TCP supports unicast delivery of data. Anycast applications exist that successfully use TCP without modifications, though there is some

risk of instability due to changes of lower-layer forwarding behavior [45].

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

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

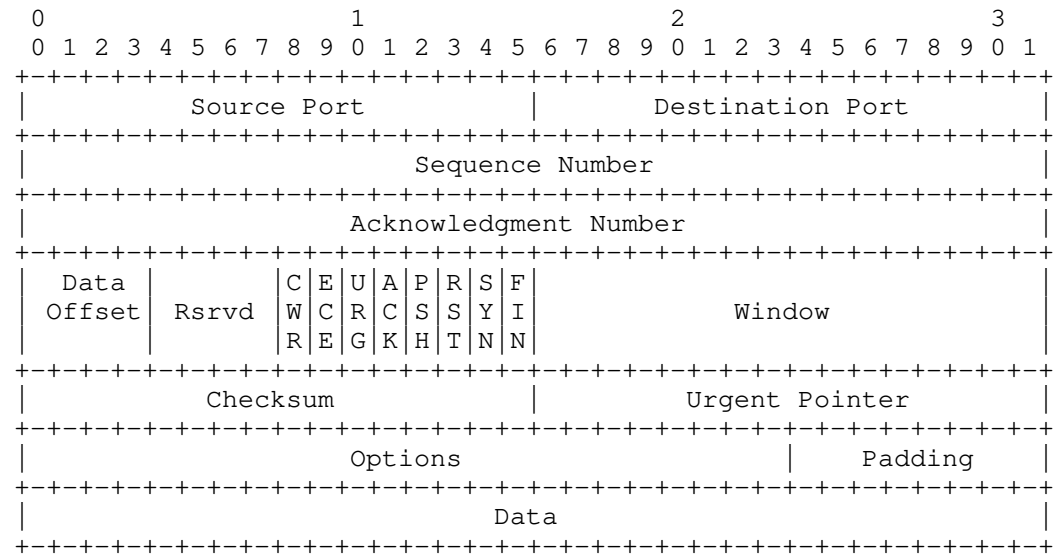
TCP uses port numbers to identify application services and to multiplex distinct flows between hosts.

A more detailed description of TCP features compared to other transport protocols can be found in Section 3.1 of [51]. Further description of the motivations for developing TCP and its role in the Internet protocol stack can be found in Section 2 of [16] and earlier versions of the TCP specification.

3. Functional Specification

3.1. Header Format

TCP segments are sent as internet datagrams. The Internet Protocol (IP) header carries several information fields, including the source and destination host addresses [1] [14]. A TCP header follows the IP headers, supplying information specific to the TCP protocol. This division allows for the existence of host level protocols other than TCP. In early development of the Internet suite of protocols, the IP header fields had been a part of TCP.



Note that one tick mark represents one bit position.

Figure 1: TCP Header Format

Each of the TCP header fields is described as follows:

Source Port: 16 bits

The source port number.

Destination Port: 16 bits

The destination port number.

Sequence Number: 32 bits

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

Acknowledgment Number: 32 bits

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

Data Offset: 4 bits

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

Rsrvd - Reserved: 4 bits

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

The control bits are also know as "flags". Assignment is managed by IANA from the "TCP Header Flags" registry [56].

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

CWR: Congestion Window Reduced (see [9])
ECE: ECN-Echo (see [9])
URG: Urgent Pointer field significant
ACK: Acknowledgment field significant
PSH: Push Function (see the Send Call description in Section 3.8.1)
RST: Reset the connection
SYN: Synchronize sequence numbers
FIN: No more data from sender

Window: 16 bits

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

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

Checksum: 16 bits

The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header and text. The checksum computation needs to ensure the 16-bit alignment of the data being summed. If a segment contains an odd number of header and text octets, alignment can be achieved by padding the last octet with zeros on its right to form a 16 bit word for checksum purposes.

The pad is not transmitted as part of the segment. While computing the checksum, the checksum field itself is replaced with zeros.

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

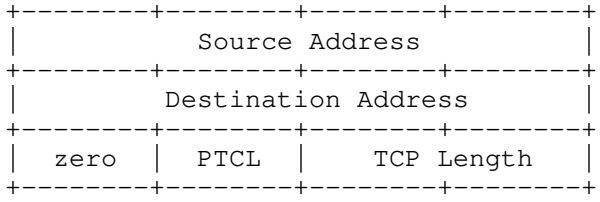


Figure 2: IPv4 Pseudo Header

Pseudo header components:

Source Address: the IPv4 source address in network byte order

Destination Address: the IPv4 destination address in network byte order

zero: bits set to zero

PTCL: the protocol number from the IP header

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

For IPv6, the pseudo header is defined in Section 8.1 of RFC 8200 [14], and contains the IPv6 Source Address and Destination Address, an Upper Layer Packet Length (a 32-bit value otherwise equivalent to TCP Length in the IPv4 pseudo header), three bytes of zero-padding, and a Next Header value (differing from the IPv6 header value in the case of extension headers present in between IPv6 and TCP).

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

Urgent Pointer: 16 bits

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

Options: variable

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

Case 1: A single octet of option-kind.

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

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

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

The list of all currently defined options is managed by IANA [55], and each option is defined in other RFCs, as indicated there. That set includes experimental options that can be extended to support multiple concurrent usages [44].

A given TCP implementation can support any currently defined options, but the following options MUST be supported (MUST-4 - note Maximum Segment Size option support is also part of MUST-19 in Section 3.6.2):

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

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

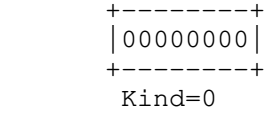
A TCP implementation MUST (MUST-6) ignore without error any TCP option it does not implement, assuming that the option has a length

field. All TCP options except End of option list and No-Operation MUST have length fields, including all future options (MUST-68). TCP implementations MUST be prepared to handle an illegal option length (e.g., zero); a suggested procedure is to reset the connection and log the error cause (MUST-7).

Note: There is ongoing work to extend the space available for TCP options, such as [60].

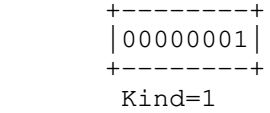
Specific Option Definitions

End of Option List



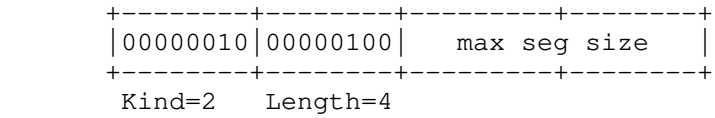
This option code indicates the end of the option list. This might not coincide with the end of the TCP header according to the Data Offset field. This is used at the end of all options, not the end of each option, and need only be used if the end of the options would not otherwise coincide with the end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

No-Operation



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

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 endpoint that sends this segment. This value is limited by the IP reassembly limit. This field may be sent in the initial connection request (i.e., in segments with the SYN control bit set) and MUST NOT be sent in other segments (MUST-65). If this option is not used, any segment size is allowed. A more complete description of this option is provided in Section 3.6.1.

Other Common Options

Additional RFCs define some other commonly used options that are recommended to implement for high performance, but not necessary for basic TCP interoperability. These are the TCP Selective Acknowledgement (SACK) option [21][24], TCP Timestamp (TS) option [46], and TCP Window Scaling (WS) option [46].

Experimental TCP Options

Experimental TCP option values are defined in [27], and [44] describes the current recommended usage for these experimental values.

Padding: variable

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

3.2. TCP Terminology Overview

This section includes an overview of key terms needed to understand the detailed protocol operation in the rest of the document. There is a traditional glossary of terms in Section 3.10.

3.2.1. Key Connection State Variables

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

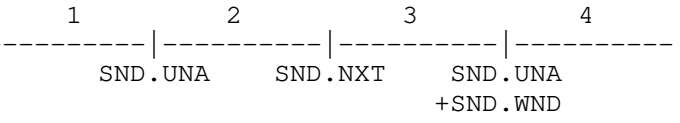
Send Sequence Variables:

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

Receive Sequence Variables:

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

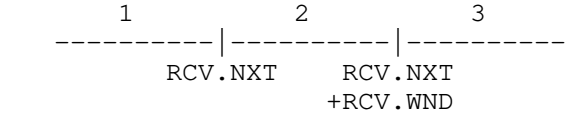
The following diagrams may help to relate some of these variables to the sequence space.



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

Figure 3: Send Sequence Space

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



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

Figure 4: Receive Sequence Space

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

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

Current Segment Variables:

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

3.2.2. State Machine Overview

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

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

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

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

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

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

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

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

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

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

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

CLOSED - represents no connection state at all.

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

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

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

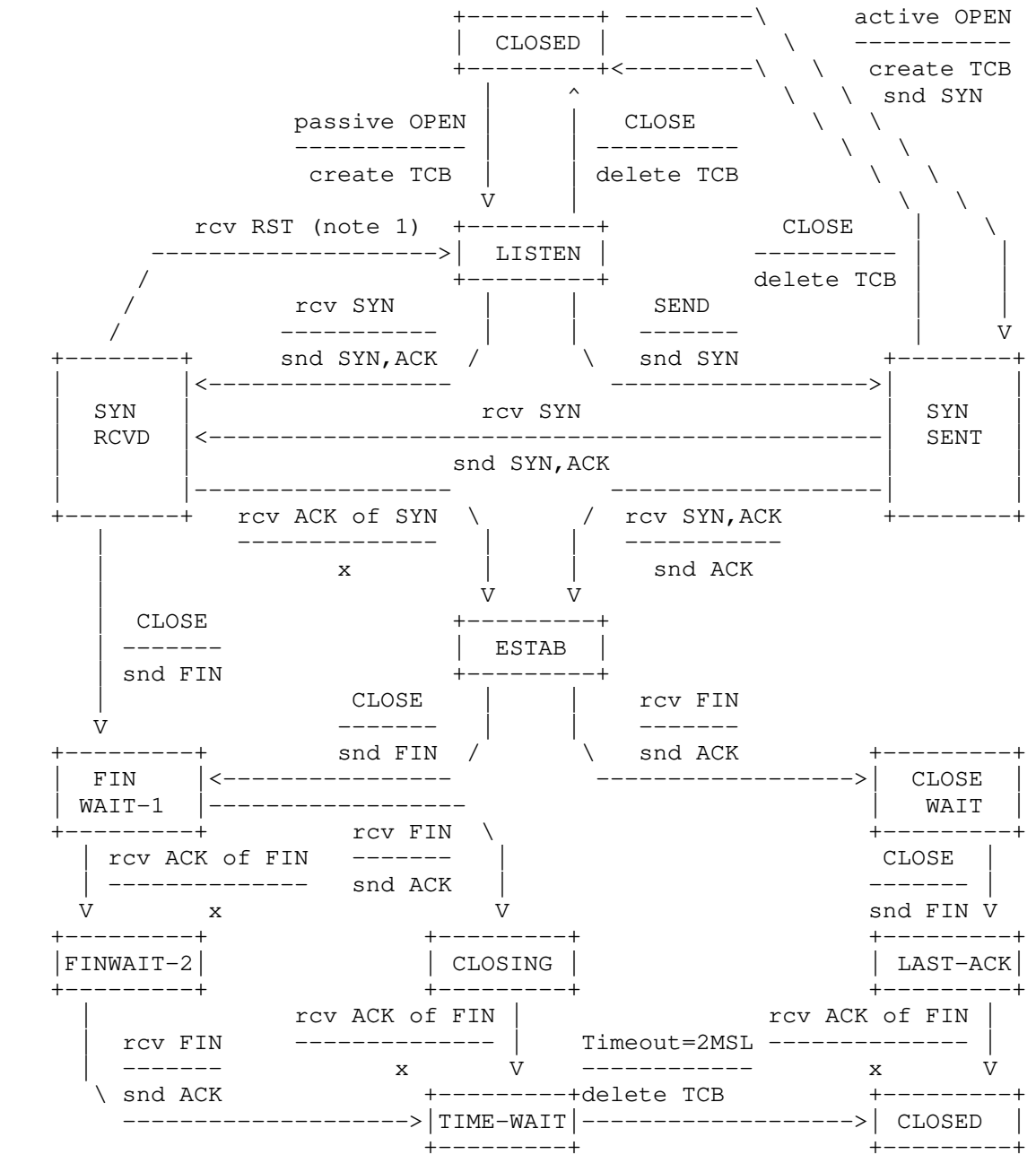


Figure 5: TCP Connection State Diagram

The following notes apply to Figure 5:

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

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

Note 3: A RST can be sent from any state with a corresponding transition to TIME-WAIT (see [63] for rationale). These transitions are not not explicitly shown, otherwise the diagram would become very difficult to read. Similarly, receipt of a RST from any state results in a transition to LISTEN or CLOSED, though this is also omitted from the diagram for legibility.

3.3. Sequence Numbers

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

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

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

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

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

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

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

SND.UNA = oldest unacknowledged sequence number

SND.NXT = next sequence number to be sent

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

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 possible for a TCP implementation to maintain a zero receive window while transmitting data and receiving ACKs. A TCP receiver MUST process the RST and URG fields of all incoming segments, even when the receive window is zero (MUST-66).

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

Initial Sequence Number Selection

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

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

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

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

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

where M is the 4 microsecond timer, and F() is a pseudorandom function (PRF) of the connection's identifying parameters ("localip, localport, remoteip, remoteport") and a secret key ("secretkey") (SHLD-1). F() MUST NOT be computable from the outside (MUST-9), or an attacker could still guess at sequence numbers from the ISN used for some other connection. The PRF could be implemented as a cryptographic hash of the concatenation of the TCP connection parameters and some secret data. For discussion of the selection of

a specific hash algorithm and management of the secret key data, please see Section 3 of [41].

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

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

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

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

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

A 3WHS is necessary because sequence numbers are not tied to a global clock in the network, and TCP implementations may have different mechanisms for picking the ISNs. The receiver of the first SYN has no way of knowing whether the segment was an old 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 [62].

Knowing When to Keep Quiet

A theoretical problem exists where data could be corrupted due to confusion between old segments in the network and new ones after a host reboots, if the same port numbers and sequence space are reused. The "Quiet Time" concept discussed below addresses this and the discussion of it is included for situations where it might be relevant, although it is not felt to be necessary in most current implementations. The problem was more relevant earlier in the

history of TCP. In practical use on the Internet today, the error-prone conditions are sufficiently unlikely that it is felt safe to ignore. Reasons why it is now negligible include: (a) ISS and ephemeral port randomization have reduced likelihood of reuse of port numbers and sequence numbers after reboots, (b) the effective MSL of the Internet has declined as links have become faster, and (c) reboots often taking longer than an MSL anyways.

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

The TCP Quiet Time Concept

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

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

Under normal conditions, TCP implementations keep track of the next sequence number to emit and the oldest awaiting acknowledgment so as to avoid mistakenly using a sequence number over before its first use

has been acknowledged. This alone does not guarantee that old duplicate data is drained from the net, so the sequence space has been made very large to reduce the probability that a wandering duplicate will cause trouble upon arrival. At 2 megabits/sec. it takes 4.5 hours to use up 2**32 octets of sequence space. Since the maximum segment lifetime in the net is not likely to exceed a few tens of seconds, this is deemed ample protection for foreseeable nets, even if data rates escalate to l0's of megabits/sec. At 100 megabits/sec, the cycle time is 5.4 minutes, which may be a little short, but still within reason.

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

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

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

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

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

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

3.4. Establishing a connection

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

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

The simplest 3WHS is shown in Figure 6. The figures should be interpreted in the following way. Each line is numbered for reference purposes. Right arrows (-->) indicate departure of a TCP segment from TCP peer A to TCP peer B, or arrival of a segment at B from A. Left arrows (<--), indicate the reverse. Ellipsis (...)

indicates a segment that is still in the network (delayed). Comments appear in parentheses. TCP connection states represent the state AFTER the departure or arrival of the segment (whose contents are shown in the center of each line). Segment contents are shown in abbreviated form, with sequence number, control flags, and ACK field. Other fields such as window, addresses, lengths, and text have been left out in the interest of clarity.

```
    TCP Peer A                                                TCP Peer B

1.  CLOSED                                                    LISTEN

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

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

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

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

Figure 6: Basic 3-Way Handshake for Connection Synchronization

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

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

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

```
    TCP Peer A                                                TCP Peer B

1.  CLOSED                                                    CLOSED

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

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

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

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

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

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

Figure 7: Simultaneous Connection Synchronization

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

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

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

```
      TCP Peer A                                                TCP Peer B

   1. CLOSED                                                    LISTEN

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

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

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

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

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

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

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

Figure 8: Recovery from Old Duplicate SYN

As a simple example of recovery from old duplicates, consider Figure 8. At line 3, an old duplicate SYN arrives at TCP Peer B. TCP Peer B cannot tell that this is an old duplicate, so it responds normally (line 4). TCP Peer A detects that the ACK field is incorrect and returns a RST (reset) with its SEQ field selected to make the segment believable. TCP Peer B, on receiving the RST, returns to the LISTEN state. When the original SYN finally arrives at line 6, the synchronization proceeds normally. If the SYN at line 6 had arrived before the RST, a more complex exchange might have occurred with RST's sent in both directions.

Half-Open Connections and Other Anomalies

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

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

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

```
      TCP Peer A                                          TCP Peer B

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

  2.  CLOSED                                               ESTABLISHED

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

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

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

  6.  SYN-SENT                                             CLOSED

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

Figure 9: Half-Open Connection Discovery

When the SYN arrives at line 3, TCP Peer B, being in a synchronized state, and the incoming segment outside the window, responds with an acknowledgment indicating what sequence it next expects to hear (ACK 100). TCP Peer A sees that this segment does not acknowledge anything it sent and, being unsynchronized, sends a reset (RST) because it has detected a half-open connection. TCP Peer B aborts at line 5. TCP Peer A will continue to try to establish the connection; the problem is now reduced to the basic 3-way handshake of Figure 6.

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

RST is acceptable so TCP Peer B processes it and aborts the connection.

```
      TCP Peer A                                          TCP Peer B

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

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

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

Figure 10: Active Side Causes Half-Open Connection Discovery

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

```
    TCP Peer A                                          TCP Peer B

1.  LISTEN                                              LISTEN

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

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

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

5.  LISTEN                                              LISTEN
```

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

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

Reset Generation

A TCP user or application can issue a reset on a connection at any time, though reset events are also generated by the protocol itself when various error conditions occur, as described below. The side of a connection issuing a reset should enter the TIME-WAIT state, as this generally helps to reduce the load on busy servers for reasons described in [63].

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

There are three groups of states:

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

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

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

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

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

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

Reset Processing

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

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

TCP implementations SHOULD allow a received RST segment to include data (SHLD-2).

3.5. Closing a Connection

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

There are essentially three cases:

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

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

3) Both users CLOSE simultaneously (Figure 13).

Case 1: Local user initiates the close

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

Case 2: TCP endpoint receives a FIN from the network

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

Case 3: Both users close simultaneously

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

```
    TCP Peer A                                              TCP Peer B

1.  ESTABLISHED                                             ESTABLISHED

2.  (Close)
    FIN-WAIT-1  --> <SEQ=100><ACK=300><CTL=FIN,ACK>  --> CLOSE-WAIT

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

4.                                                          (Close)
    TIME-WAIT   <-- <SEQ=300><ACK=101><CTL=FIN,ACK>  <-- LAST-ACK

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

6.  (2 MSL)
    CLOSED
```

Figure 12: Normal Close Sequence

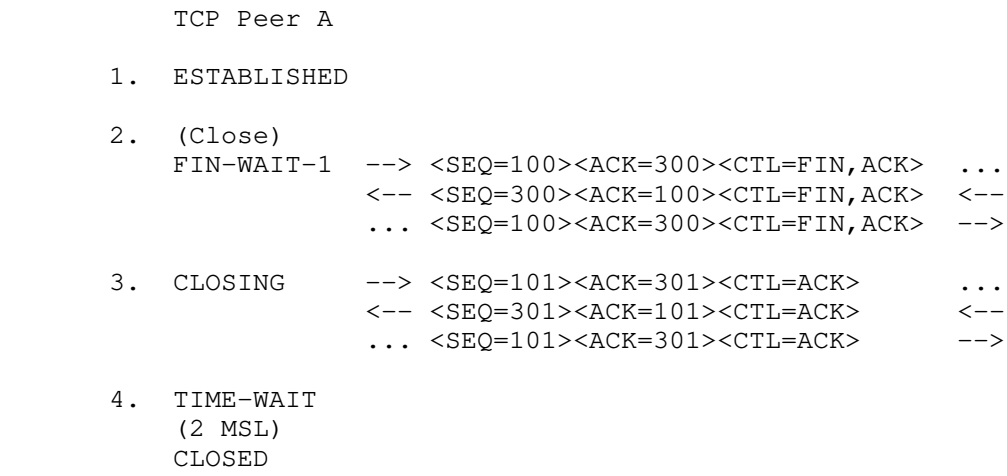


Figure 13: Simultaneous Close Sequence

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

3.5.1. Half-Closed Connections

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

A host MAY implement a "half-duplex" TCP close sequence, so that an application that has called CLOSE cannot continue to read data from the connection (MAY-1). If such a host issues a CLOSE call while received data is still pending in the TCP connection, or if new data is received after CLOSE is called, its TCP implementation SHOULD send a RST to show that data was lost (SHLD-3). See [22] section 2.17 for discussion.

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

(1) assigns its initial sequence number for the new connection to be larger than the largest sequence number it used on the previous connection incarnation, and

(2) returns to TIME-WAIT state if the SYN turns out to be an old duplicate.

When the TCP Timestamp options are available, an improved algorithm is described in [39] in order to support higher connection establishment rates. This algorithm for reducing TIME-WAIT is a Best Current Practice that SHOULD be implemented, since timestamp options are commonly used, and using them to reduce TIME-WAIT provides benefits for busy Internet servers (SHLD-4).

3.6. Segmentation

The term "segmentation" refers to the activity TCP performs when ingesting a stream of bytes from a sending application and packetizing that stream of bytes into TCP segments. Individual TCP segments often do not correspond one-for-one to individual send (or socket write) calls from the application. Applications may perform writes at the granularity of messages in the upper layer protocol, but TCP guarantees no boundary coherence between the TCP segments sent and received versus user application data read or write buffer boundaries. In some specific protocols, such as Remote Direct Memory Access (RDMA) using Direct Data Placement (DDP) and Marker PDU Aligned Framing (MPA) [31], there are performance optimizations possible when the relation between TCP segments and application data units can be controlled, and MPA includes a specific mechanism for detecting and verifying this relationship between TCP segments and application message data structures, but this is specific to applications like RDMA. In general, multiple goals influence the sizing of TCP segments created by a TCP implementation.

Goals driving the sending of larger segments include:

- Reducing the number of packets in flight within the network.
- Increasing processing efficiency and potential performance by enabling a smaller number of interrupts and inter-layer interactions.
- Limiting the overhead of TCP headers.

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

performance than 1024 bytes, due purely to data alignment on copy operations.

Goals driving the sending of smaller segments include:

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

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

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

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

3.6.1. Maximum Segment Size Option

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

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

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

The maximum size of a segment that TCP endpoint really sends, the "effective send MSS," MUST be the smaller (MUST-16) of the send MSS (that reflects the available reassembly buffer size at the remote host, the EMTU_R [18]) and the largest transmission size permitted by the IP layer (EMTU_S [18]):

```
    Eff.snd.MSS =

        min(SendMSS+20, MMS_S) - TCPhdrsize - IPoptionsize
```

where:

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

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

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

- IPoptionsize is the size of any IP options associated with a TCP connection. Note that some options may not be included on all packets, but that for each segment sent, the sender should adjust the data length accordingly, within the Eff.snd.MSS.

The MSS value to be sent in an MSS option should be equal to the effective MTU minus the fixed IP and TCP headers. By ignoring both IP and TCP options when calculating the value for the MSS option, if there are any IP or TCP options to be sent in a packet, then the sender must decrease the size of the TCP data accordingly. RFC 6691 [42] discusses this in greater detail.

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

```
MMS_R - 20
```

where MMS_R is the maximum size for a transport-layer message that can be received (and reassembled at the IP layer) (MUST-67). TCP obtains MMS_R and MMS_S from the IP layer; see the generic call GET_MAXSIZES in Section 3.4 of RFC 1122. These are defined in terms of their IP MTU equivalents, EMTU_R and EMTU_S [18].

When TCP is used in a situation where either the IP or TCP headers are not fixed, the sender must reduce the amount of TCP data in any given packet by the number of octets used by the IP and TCP options. This has been a point of confusion historically, as explained in RFC 6691, Section 3.1.

3.6.2. Path MTU Discovery

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

PMTUD for IPv4 [2] or IPv6 [3] is implemented in conjunction between TCP, IP, and ICMP protocols. It relies both on avoiding source fragmentation and setting the IPv4 DF (don't fragment) flag, the latter to inhibit on-path fragmentation. It relies on ICMP errors from routers along the path, whenever a segment is too large to traverse a link. Several adjustments to a TCP implementation with PMTUD are described in RFC 2923 in order to deal with problems experienced in practice [8]. PLPMTUD [28] is a Standards Track improvement to PMTUD that relaxes the requirement for ICMP support across a path, and improves performance in cases where ICMP is not consistently conveyed, but still tries to avoid source fragmentation. The mechanisms in all four of these RFCs are recommended to be included in TCP implementations.

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

3.6.3. Interfaces with Variable MTU Values

The effective MTU can sometimes vary, as when used with variable compression, e.g., RObust Header Compression (ROHC) [35]. It is tempting for a TCP implementation to advertise the largest possible MSS, to support the most efficient use of compressed payloads. Unfortunately, some compression schemes occasionally need to transmit full headers (and thus smaller payloads) to resynchronize state at their endpoint compressors/decompressors. If the largest MTU is used to calculate the value to advertise in the MSS option, TCP retransmission may interfere with compressor resynchronization.

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

3.6.4. Nagle Algorithm

The "Nagle algorithm" was described in RFC 896 [17] and was recommended in RFC 1122 [18] for mitigation of an early problem of too many small packets being generated. It has been implemented in most current TCP code bases, sometimes with minor variations (see Appendix A.3).

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

A TCP implementation SHOULD implement the Nagle Algorithm to coalesce short segments (SHLD-7). However, there MUST be a way for an application to disable the Nagle algorithm on an individual connection (MUST-17). In all cases, sending data is also subject to the limitation imposed by the Slow Start algorithm [34].

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

3.6.5. IPv6 Jumbograms

In order to support TCP over IPv6 Jumbograms, implementations need to be able to send TCP segments larger than the 64KB limit that the MSS option can convey. RFC 2675 [6] defines that an MSS value of 65,535 bytes is to be treated as infinity, and Path MTU Discovery [3] is used to determine the actual MSS.

The Jumbo Payload option need not be implemented or understood by IPv6 nodes that do not support attachment to links with a MTU greater than 65,575 [6], and the present IPv6 Node Requirements does not include support for Jumbograms [53].

3.7. Data Communication

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

segments may arrive due to network or TCP retransmission. As discussed in the section on sequence numbers the TCP implementation performs certain tests on the sequence and acknowledgment numbers in the segments to verify their acceptability.

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

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

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

3.7.1. Retransmission Timeout

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

The RTO MUST be computed according to the algorithm in [11], including Karn's algorithm for taking RTT samples (MUST-18).

RFC 793 contains an early example procedure for computing the RTO. This was then replaced by the algorithm described in RFC 1122, and subsequently updated in RFC 2988, and then again in RFC 6298.

RFC 1122 allows that if a retransmitted packet is identical to the original packet (which implies not only that the data boundaries have not changed, but also that none of the headers have changed), then the same IPv4 Identification field MAY be used (see Section 3.2.1.5 of RFC 1122) (MAY-4). The same IP identification field may be reused anyways, since it is only meaningful when a datagram is fragmented [43]. TCP implementations should not rely on or typically interact with this IPv4 header field in any way. It is not a reasonable way to either indicate duplicate sent segments, nor to identify duplicate received segments.

3.7.2. TCP Congestion Control

RFC 2914 [7] explains the importance of congestion control for the Internet.

RFC 1122 required implementation of Van Jacobson's congestion control algorithms slow start with congestion avoidance together with exponential back-off for successive RTO values for the same segment. RFC 2581 provided IETF Standards Track description of slow start and congestion avoidance, along with fast retransmit and fast recovery. RFC 5681 is the current description of these algorithms and is the current Standards Track specification providing guidelines for TCP congestion control. RFC 6298 describes exponential back-off of RTO values, including keeping the backed-off value until a subsequent segment with new data has been sent and acknowledged.

A TCP endpoint MUST implement the basic congestion control algorithms slow start, congestion avoidance, and exponential back-off of RTO to avoid creating congestion collapse conditions (MUST-19). RFC 5681 and RFC 6298 describe the basic algorithms on the IETF Standards Track that are broadly applicable. Multiple other suitable algorithms exist and have been widely used. Many TCP implementations support a set of alternative algorithms that can be configured for use on the endpoint. An endpoint may implement such alternative algorithms provided that the algorithms are conformant with the TCP specifications from the IETF Standards Track as described in RFC 2914, RFC 5033 [10], and RFC 8961 [15].

Explicit Congestion Notification (ECN) was defined in RFC 3168 and is an IETF Standards Track enhancement that has many benefits [50].

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

3.7.3. TCP Connection Failures

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

(a) There are two thresholds R1 and R2 measuring the amount of retransmission that has occurred for the same segment. R1 and R2 might be measured in time units or as a count of retransmissions.

(b) When the number of transmissions of the same segment reaches or exceeds threshold R1, pass negative advice (see Section 3.3.1.4 of [18]) to the IP layer, to trigger dead-gateway diagnosis.

(c) When the number of transmissions of the same segment reaches a threshold R2 greater than R1, close the connection.

(d) An application MUST (MUST-21) be able to set the value for R2 for a particular connection. For example, an interactive application might set R2 to "infinity," giving the user control over when to disconnect.

(e) TCP implementations SHOULD inform the application of the delivery problem (unless such information has been disabled by the application; see Asynchronous Reports section), when R1 is reached and before R2 (SHLD-9). This will allow a remote login (User Telnet) application program to inform the user, for example.

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

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

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

3.7.4. TCP Keep-Alives

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

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

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

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

3.7.5. The Communication of Urgent Information

As a result of implementation differences and middlebox interactions, new applications SHOULD NOT employ the TCP urgent mechanism (SHLD-13). However, TCP implementations MUST still include support for the urgent mechanism (MUST-30). Details can be found in RFC 6093 [38].

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

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

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

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

A TCP implementation MUST support a sequence of urgent data of any length (MUST-31). [18]

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

A TCP implementation MUST (MUST-32) inform the application layer asynchronously whenever it receives an Urgent pointer and there was previously no pending urgent data, or whenever the Urgent pointer advances in the data stream. The TCP implementation MUST (MUST-33) provide a way for the application to learn how much urgent data remains to be read from the connection, or at least to determine whether or not more urgent data remains to be read [18].

3.7.6. Managing the Window

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

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

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

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

The mechanisms provided allow a TCP endpoint to advertise a large window and to subsequently advertise a much smaller window without having accepted that much data. This, so called "shrinking the window," is strongly discouraged. The robustness principle [18]

dictates that TCP peers will not shrink the window themselves, but will be prepared for such behavior on the part of other TCP peers.

A TCP receiver SHOULD NOT shrink the window, i.e., move the right window edge to the left (SHLD-14). However, a sending TCP peer MUST be robust against window shrinking, which may cause the "usable window" (see Section 3.7.6.2.1) to become negative (MUST-34).

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

3.7.6.1. Zero Window Probing

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

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

A TCP implementation MAY keep its offered receive window closed indefinitely (MAY-8). As long as the receiving TCP peer continues to send acknowledgments in response to the probe segments, the sending TCP peer MUST allow the connection to stay open (MUST-37). This enables TCP to function in scenarios such as the "printer ran out of paper" situation described in Section 4.2.2.17 of RFC1122. The behavior is subject to the implementation’s resource management concerns, as noted in [40].

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

The transmitting host SHOULD send the first zero-window probe when a zero window has existed for the retransmission timeout period (SHLD-29) (Section 3.7.1), and SHOULD increase exponentially the interval between successive probes (SHLD-30).

3.7.6.2. Silly Window Syndrome Avoidance

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

3.7.6.2.1. Sender's Algorithm - When to Send Data

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

The Nagle algorithm from Section 3.6.4 additionally describes how to coalesce short segments.

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

The "usable window" is:

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

i.e., the offered window less the amount of data sent but not acknowledged. If D is the amount of data queued in the sending TCP endpoint but not yet sent, then the following set of rules is recommended.

Send data:

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

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

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

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

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

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

[SND.NXT = SND.UNA and]

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

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

Here Fs is a fraction whose recommended value is 1/2. The override timeout should be in the range 0.1 - 1.0 seconds. It may be convenient to combine this timer with the timer used to probe zero windows (Section 3.7.6.1).

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

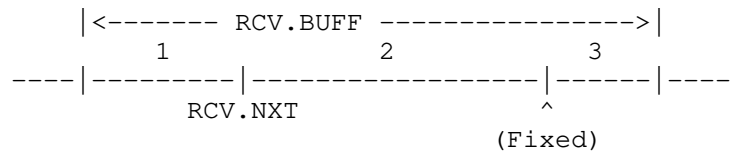
A TCP implementation MUST include a SWS avoidance algorithm in the receiver (MUST-39).

The receiver's SWS avoidance algorithm determines when the right window edge may be advanced; this is customarily known as "updating the window". This algorithm combines with the delayed ACK algorithm (Section 3.7.6.3) to determine when an ACK segment containing the current window will really be sent to the receiver.

The solution to receiver SWS is to avoid advancing the right window edge RCV.NXT+RCV.WND in small increments, even if data is received from the network in small segments.

Suppose the total receive buffer space is RCV.BUFF. At any given moment, RCV.USER octets of this total may be tied up with data that has been received and acknowledged but that the user process has not yet consumed. When the connection is quiescent, RCV.WND = RCV.BUFF and RCV.USER = 0.

Keeping the right window edge fixed as data arrives and is acknowledged requires that the receiver offer less than its full buffer space, i.e., the receiver must specify a RCV.WND that keeps RCV.NXT+RCV.WND constant as RCV.NXT increases. Thus, the total buffer space RCV.BUFF is generally divided into three parts:



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

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

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

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

where Fr is a fraction whose recommended value is 1/2, and Eff.snd.MSS is the effective send MSS for the connection (see Section 3.6.1). When the inequality is satisfied, RCV.WND is set to RCV.BUFF-RCV.USER.

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

3.7.6.3. Delayed Acknowledgements - When to Send an ACK Segment

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

A TCP endpoint SHOULD implement a delayed ACK (SHLD-18), but an ACK should not be excessively delayed; in particular, the delay MUST be less than 0.5 seconds (MUST-40), and in a stream of full-sized segments there SHOULD be an ACK for at least every second segment (SHLD-19). Excessive delays on ACKs can disturb the round-trip timing and packet "clocking" algorithms. More complete discussion of delayed ACK behavior is in Section 4.2 of RFC 5681 [34], including rules for streams of segments that are not full-sized. Note that there are several current practices that further lead to a reduced number of ACKs, including generic receive offload (GRO), ACK compression, and ACK decimation [25].

3.8. Interfaces

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

3.8.1. User/TCP Interface

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

Section 3.1 of [52] also identifies primitives provided by TCP, and could be used as an additional reference for implementers.

TCP User Commands

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

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

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

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

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

Open

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Send

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

This call causes the data contained in the indicated user buffer to be sent on the indicated connection. If the

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

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

If the PUSH flag is set, the application intends the data to be transmitted promptly to the receiver, and the PUSH bit will be set in the last TCP segment created from the buffer. When an application issues a series of SEND calls without setting the PUSH flag, the TCP implementation MAY aggregate the data internally without sending it (MAY-16).

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

If the PUSH flag is not set, the data may be combined with data from subsequent SENDs for transmission efficiency. Note that when the Nagle algorithm is in use, TCP implementations may buffer the data before sending, without regard to the PUSH flag (see Section 3.6.4).

An application program is logically required to set the PUSH flag in a SEND call whenever it needs to force delivery of the data to avoid a communication deadlock. However, a TCP implementation SHOULD send a maximum-sized segment whenever possible (SHLD-28), to improve performance (see Section 3.7.6.2.1).

New applications SHOULD NOT set the URGENT flag [38] due to implementation differences and middlebox issues (SHLD-13).

If the URGENT flag is set, segments sent to the destination TCP peer will have the urgent pointer set. The receiving TCP peer will signal the urgent condition to the receiving process if the urgent pointer indicates that data preceding the urgent pointer has not been consumed by the receiving process. The

purpose of urgent is to stimulate the receiver to process the urgent data and to indicate to the receiver when all the currently known urgent data has been received. The number of times the sending user's TCP implementation signals urgent will not necessarily be equal to the number of times the receiving user will be notified of the presence of urgent data.

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

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

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

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

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

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

Receive

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

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

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

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

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

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

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

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

Close

Format: CLOSE (local connection name)

This command causes the connection specified to be closed. If the connection is not open or the calling process is not authorized to use this connection, an error is returned. Closing connections is intended to be a graceful operation in the sense that outstanding SENDs will be transmitted (and retransmitted), as flow control permits, until all have been serviced. Thus, it should be acceptable to make several SEND calls, followed by a CLOSE, and expect all the data to be sent to the destination. It should also be clear that users should continue to RECEIVE on CLOSING connections, since the remote peer may be trying to transmit the last of its data. Thus, CLOSE means "I have no more to send" but does not mean "I will not receive any more." It may happen (if the user level protocol is not well thought out) that the closing side is unable to get rid of all its data before timing out. In this event, CLOSE turns into ABORT, and the closing TCP peer gives up.

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

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

Close also implies push function.

Status

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

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

This command returns a data block containing the following information:

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

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

Abort

Format: ABORT (local connection name)

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

Flush

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

Asynchronous Reports

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

ERROR_REPORT(local connection name, reason, subreason)

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

* ICMP error message arrived (see Section 3.8.2.2 for description of handling each ICMP message type, since some message types need to be suppressed from generating reports to the application)

* Excessive retransmissions (see Section 3.7.3)

* Urgent pointer advance (see Section 3.7.5)

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

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

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

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

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

3.8.2. TCP/Lower-Level Interface

The TCP endpoint calls on a lower level protocol module to actually send and receive information over a network. The two current standard Internet Protocol (IP) versions layered below TCP are IPv4 [1] and IPv6 [14].

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

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

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

Note that RFC 793 specified one minute (60 seconds) as a constant for the TTL, because the assumed maximum segment lifetime was two minutes. This was intended to explicitly ask that a segment be destroyed if it cannot be delivered by the internet system within one minute. RFC 1122 changed this specification to require that the TTL be configurable.

Note that the DiffServ field is permitted to change during a connection (Section 4.2.4.2 of RFC 1122). However, the application interface might not support this ability, and the application does not have knowledge about individual TCP segments, so this can only be done on a coarse granularity, at best. This limitation is further discussed in RFC 7657 (sec 5.1, 5.3, and 6) [49]. Generally, an application SHOULD NOT change the DiffServ field value during the course of a connection (SHLD-23).

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

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

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

3.8.2.1. Source Routing

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

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

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

3.8.2.2. ICMP Messages

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

This applies to ICMPv6 in addition to IPv4 ICMP.

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

Source Quench
: TCP implementations MUST silently discard any received ICMP Source Quench messages (MUST-55). See [12] for discussion.

Soft Errors
: For ICMP these include: Destination Unreachable -- codes 0, 1, 5, Time Exceeded -- codes 0, 1, and Parameter Problem.
For ICMPv6 these include: Destination Unreachable -- codes 0 and 3, Time Exceeded -- codes 0, 1, and Parameter Problem -- codes 0, 1, 2
Since these Unreachable messages indicate soft error conditions, TCP implementations MUST NOT abort the connection (MUST-56), and it SHOULD make the information available to the application (SHLD-25).

Hard Errors

> For ICMP these include Destination Unreachable -- codes 2-4">
> These are hard error conditions, so TCP implementations SHOULD abort the connection (SHLD-26). [32] notes that some implementations do not abort connections when an ICMP hard error is received for a connection that is in any of the synchronized states.

Note that [32] section 4 describes widespread implementation behavior that treats soft errors as hard errors during connection establishment.

3.8.2.3. Source Address Validation

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

> An incoming SYN with an invalid source address MUST be ignored either by TCP or by the IP layer (MUST-63) (Section 3.2.1.3 of [18]).
>
> A TCP implementation MUST silently discard an incoming SYN segment that is addressed to a broadcast or multicast address (MUST-57).

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

3.9. Event Processing

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

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

Events that occur:

> User Calls
>
> OPEN
> SEND
> RECEIVE

```
         CLOSE
         ABORT
         STATUS

      Arriving Segments

         SEGMENT ARRIVES

      Timeouts

         USER TIMEOUT
         RETRANSMISSION TIMEOUT
         TIME-WAIT TIMEOUT
```

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

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

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

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

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

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

OPEN Call

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

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

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

LISTEN STATE

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

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

SYN-SENT STATE
SYN-RECEIVED STATE

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

ESTABLISHED STATE
CLOSE-WAIT STATE

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

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

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

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

RECEIVE Call

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

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

Otherwise return "error: connection does not exist".

LISTEN STATE
SYN-SENT STATE
SYN-RECEIVED STATE

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

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

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

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

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

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

CLOSE-WAIT STATE

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

CLOSING STATE
LAST-ACK STATE

```
      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 endpoint that sent the offending segment.

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

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

If the ACK bit is on,

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

Return.

If the state is LISTEN then

first check for an RST

An incoming RST should be ignored. Return.

second check for an ACK

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

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

Return.

third check for a SYN

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

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

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

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

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

fourth other text or control

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

If the state is SYN-SENT then

first check the ACK bit

If the ACK bit is set

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

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

and discard the segment. Return.

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

second check the RST bit

If the RST bit is set

A potential blind reset attack is described in RFC 5961 [37]. The mitigation described in that document has specific applicability explained therein, and is not a substitute for cryptographic protection (e.g. IPsec or TCP-AO). A TCP implementation that supports the RFC 5961 mitigation SHOULD first check that the sequence number exactly matches RCV.NXT prior to executing the action in the next paragraph.

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

third check the security

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

If there is an ACK

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

Otherwise

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

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

fourth check the SYN bit

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

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

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

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

and send it. Data or controls that were queued for transmission may be included. If there are other controls

or text in the segment then continue processing at the sixth step below where the URG bit is checked, otherwise return.

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

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

and send it. Set the variables:

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

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

Note that it is legal to send and receive application data on SYN segments (this is the "text in the segment" mentioned above. There has been significant misinformation and misunderstanding of this topic historically. Some firewalls and security devices consider this suspicious. However, the capability was used in T/TCP [20] and is used in TCP Fast Open (TFO) [47], so is important for implementations and network devices to permit.

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

Otherwise,

first check sequence number

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

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

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

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

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

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

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

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

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

Note that for the TIME-WAIT state, there is an improved algorithm described in [39] for handling incoming SYN segments, that utilizes timestamps rather than relying on the sequence number check described here. When the improved algorithm is implemented, the logic above is not applicable for incoming SYN segments with timestamp options, received on a connection in the TIME-WAIT state.

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

second check the RST bit,

RFC 5961 [37] section 3 describes a potential blind reset attack and optional mitigation approach. This does not provide a cryptographic protection (e.g. as in IPsec or TCP-AO), but can be applicable in situations described in RFC 5961. For stacks implementing the RFC 5961 protection, the three checks below apply, otherwise processing for these states is indicated further below.

1) If the RST bit is set and the sequence number is outside the current receive window, silently drop the segment.

2) If the RST bit is set and the sequence number exactly matches the next expected sequence number (RCV.NXT), then TCP endpoints MUST reset the connection in the manner prescribed below according to the connection state.

3) If the RST bit is set and the sequence number does not exactly match the next expected sequence value, yet is within the current receive window, TCP endpoints MUST send an acknowledgement (challenge ACK):

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

After sending the challenge ACK, TCP endpoints MUST drop the unacceptable segment and stop processing the incoming packet further. Note that RFC 5961 and Errata ID 4772 contain additional considerations for ACK throttling in an implementation.

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

SYN-RECEIVED

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

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

If the security/compartment in the segment does not exactly match the security/compartment in the TCB then send a reset, any outstanding RECEIVEs and SEND should receive "reset" responses. All segment queues should be flushed. Users should also receive an unsolicited

general "connection reset" signal. Enter the CLOSED state, delete the TCB, and return.

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

fourth, check the SYN bit,

SYN-RECEIVED

If the connection was initiated with a passive OPEN, then return this connection to the LISTEN state and return. Otherwise, handle per the directions for synchronized states below.

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

If the SYN bit is set in these synchronized states, it may be either a legitimate new connection attempt (e.g. in the case of TIME-WAIT), an error where the connection should be reset, or the result of an attack attempt, as described in RFC 5961 [37]. For the TIME-WAIT state, new connections can be accepted if the timestamp option is used and meets expectations (per [39]). For all other cases, RFC 5961 provides a mitigation with applicability to some situations, though there are also alternatives that offer cryptographic protection (see Section 6). RFC 5961 recommends that in these synchronized states, if the SYN bit is set, irrespective of the sequence number, TCP endpoints MUST send a "challenge ACK" to the remote peer:

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

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

For implementations that do not follow RFC 5961, the original RFC 793 behavior follows in this paragraph. If the SYN is in the window it is an error, send a reset, any outstanding RECEIVEs and SEND should receive "reset" responses, all segment queues should be flushed, the user should also receive an unsolicited general "connection reset" signal, enter the CLOSED state, delete the TCB, and return.

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

fifth check the ACK field,

if the ACK bit is off drop the segment and return

if the ACK bit is on

RFC 5961 [37] section 5 describes a potential blind data injection attack, and mitigation that implementations MAY choose to include (MAY-12). TCP stacks that implement RFC 5961 MUST add an input check that the ACK value is acceptable only if it is in the range of ((SND.UNA - MAX.SND.WND) =< SEG.ACK =< SND.NXT). All incoming segments whose ACK value doesn't satisfy the above condition MUST be discarded and an ACK sent back. The new state variable MAX.SND.WND is defined as the largest window that the local sender has ever received from its peer (subject to window scaling) or may be hard-coded to a maximum permissible window value. When the ACK value is acceptable, the processing per-state below applies:

SYN-RECEIVED STATE

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

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

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

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

and send it.

ESTABLISHED STATE

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

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

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

FIN-WAIT-1 STATE

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

FIN-WAIT-2 STATE

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

CLOSE-WAIT STATE

Do the same processing as for the ESTABLISHED state.

CLOSING STATE

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

LAST-ACK STATE

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

TIME-WAIT STATE

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

sixth, check the URG bit,

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

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

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

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

seventh, process the segment text,

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

Once in the ESTABLISHED state, it is possible to deliver segment text to user RECEIVE buffers. Text from segments can be moved into buffers until either the buffer is full or the segment is empty. If the segment empties and

carries a PUSH flag, then the user is informed, when the buffer is returned, that a PUSH has been received.

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

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

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

Please note the window management suggestions in Section 3.7.

Send an acknowledgment of the form:

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

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

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

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

eighth, check the FIN bit,

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

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

SYN-RECEIVED STATE
ESTABLISHED STATE

Enter the CLOSE-WAIT state.

FIN-WAIT-1 STATE

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.

3.10. Glossary

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

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

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

Destination Address
: The network layer address of the remote endpoint.

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

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

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

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

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

internet address
: A network layer address.

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

internet fragment

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

IP
: Internet Protocol. See [1] and [14].

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

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

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

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

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

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

octet
: An eight bit byte.

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

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

port

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

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

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

RCV.NXT
: receive next sequence number

RCV.UP
: receive urgent pointer

RCV.WND
: receive window

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

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

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

SEG.ACK
: segment acknowledgment

SEG.LEN

segment length

SEG.SEQ
segment sequence

SEG.UP
segment urgent pointer field

SEG.WND
segment window field

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

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

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

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

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

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

SND.NXT
send sequence

SND.UNA
left sequence

SND.UP
: send urgent pointer

SND.WL1
: segment sequence number at last window update

SND.WL2
: segment acknowledgment number at last window update

SND.WND
: send window

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

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

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

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

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

TOS
: Type of Service, an obsoleted IPv4 field. The same header bits currently are used for the Differentiated Services field [5] containing the Differentiated Services Code Point (DSCP) value and the 2-bit ECN codepoint [9].

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

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

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

4. Changes from RFC 793

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

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

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

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

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

RFC 1122 contains a collection of other changes and clarifications to RFC 793. The normative items impacting the protocol have been incorporated here, though some historically useful implementation advice and informative discussion from RFC 1122 is not included here.

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

The more secure Initial Sequence Number generation algorithm from RFC 6528 was incorporated. See RFC 6528 for discussion of the attacks that this mitigates, as well as advice on selecting PRF algorithms and managing secret key data.

A note based on RFC 6429 was added to explicitly clarify that system resource management concerns allow connection resources to be reclaimed. RFC 6429 is obsoleted in the sense that this clarification has been reflected in this update to the base TCP specification now.

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

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

This document started as draft-eddy-rfc793bis-00, that was merely a proposal and rough plan for updating RFC 793.

The -01 revision of this draft-eddy-rfc793bis incorporates the content of RFC 793 Section 3 titled "FUNCTIONAL SPECIFICATION". Other content from RFC 793 has not been incorporated. The -01 revision of this document makes some minor formatting changes to the RFC 793 content in order to convert the content into XML2RFC format and account for left-out parts of RFC 793. For instance, figure numbering differs and some indentation is not exactly the same.

The -02 revision of draft-eddy-rfc793bis incorporates errata that have been verified:

Errata ID 573: Reported by Bob Braden (note: This errata basically is just a reminder that RFC 1122 updates 793. Some of the associated changes are left pending to a separate revision that incorporates 1122. Bob's mention of PUSH in 793 section 2.8 was not applicable here because that section was not part of the "functional specification". Also the 1122 text on the retransmission timeout also has been updated by subsequent RFCs, so the change here deviates from Bob's suggestion to apply the 1122 text.)
Errata ID 574: Reported by Yin Shuming
Errata ID 700: Reported by Yin Shuming
Errata ID 701: Reported by Yin Shuming
Errata ID 1283: Reported by Pei-chun Cheng
Errata ID 1561: Reported by Constantin Hagemeier
Errata ID 1562: Reported by Constantin Hagemeier

Errata ID 1564: Reported by Constantin Hagemeier
Errata ID 1565: Reported by Constantin Hagemeier
Errata ID 1571: Reported by Constantin Hagemeier
Errata ID 1572: Reported by Constantin Hagemeier
Errata ID 2296: Reported by Vishwas Manral
Errata ID 2297: Reported by Vishwas Manral
Errata ID 2298: Reported by Vishwas Manral
Errata ID 2748: Reported by Mykyta Yevstifeyev
Errata ID 2749: Reported by Mykyta Yevstifeyev
Errata ID 2934: Reported by Constantin Hagemeier
Errata ID 3213: Reported by EugnJun Yi
Errata ID 3300: Reported by Botong Huang
Errata ID 3301: Reported by Botong Huang
Errata ID 3305: Reported by Botong Huang
Note: Some verified errata were not used in this update, as they relate to sections of RFC 793 elided from this document. These include Errata ID 572, 575, and 1569.
Note: Errata ID 3602 was not applied in this revision as it is duplicative of the 1122 corrections.

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

The -03 revision of draft-eddy-rfc793bis revises all discussion of the urgent pointer in order to comply with RFC 6093, 1122, and 1011. Since 1122 held requirements on the urgent pointer, the full list of requirements was brought into an appendix of this document, so that it can be updated as-needed.

The -04 revision of draft-eddy-rfc793bis includes the ISN generation changes from RFC 6528.

The -05 revision of draft-eddy-rfc793bis incorporates MSS requirements and definitions from RFC 879, 1122, and 6691, as well as option-handling requirements from RFC 1122.

The -00 revision of draft-ietf-tcpm-rfc793bis incorporates several additional clarifications and updates to the section on segmentation, many of which are based on feedback from Joe Touch improving from the initial text on this in the previous revision.

The -01 revision incorporates the change to Reserved bits due to ECN, as well as many other changes that come from RFC 1122.

The -02 revision has small formatting modifications in order to address xml2rfc warnings about long lines. It was a quick update to

avoid document expiration. TCPM working group discussion in 2015 also indicated that that we should not try to add sections on implementation advice or similar non-normative information.

The -03 revision incorporates more content from RFC 1122: Passive OPEN Calls, Time-To-Live, Multihoming, IP Options, ICMP messages, Data Communications, When to Send Data, When to Send a Window Update, Managing the Window, Probing Zero Windows, When to Send an ACK Segment. The section on data communications was re-organized into clearer subsections (previously headings were embedded in the 793 text), and windows management advice from 793 was removed (as reviewed by TCPM working group) in favor of the 1122 additions on SWS, ZWP, and related topics.

The -04 revision includes reference to RFC 6429 on the ZWP condition, RFC1122 material on TCP Connection Failures, TCP Keep-Alives, Acknowledging Queued Segments, and Remote Address Validation. RTO computation is referenced from RFC 6298 rather than RFC 1122.

The -05 revision includes the requirement to implement TCP congestion control with recommendation to implement ECN, the RFC 6633 update to 1122, which changed the requirement on responding to source quench ICMP messages, and discussion of ICMP (and ICMPv6) soft and hard errors per RFC 5461 (ICMPv6 handling for TCP doesn't seem to be mentioned elsewhere in standards track).

The -06 revision includes an appendix on "Other Implementation Notes" to capture widely-deployed fundamental features that are not contained in the RFC series yet. It also added mention of RFC 6994 and the IANA TCP parameters registry as a reference. It includes references to RFC 5961 in appropriate places. The references to TOS were changed to DiffServ field, based on reflecting RFC 2474 as well as the IPv6 presence of traffic class (carrying DiffServ field) rather than TOS.

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

The -08 revision includes changes based on:

describing treatment of reserved bits (following TCPM mailing list thread from July 2014 on "793bis item - reserved bit behavior"
addition a brief TCP key concepts section to make up for not including the outdated section 2 of RFC 793
changed "TCP" to "host" to resolve conflict between 1122 wording on whether TCP or the network layer chooses an address when multihomed

fixed/updated definition of options in glossary
moved note on aggregating ACKs from 1122 to a more appropriate location
resolved notes on IP precedence and security/compartment
added implementation note on sequence number validation
added note that PUSH does not apply when Nagle is active
added 1122 content on asynchronous reports to replace 793 section on TCP to user messages

The -09 revision fixes section numbering problems.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5. IANA Considerations

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

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

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

IANA should add a column for "Assignment Notes".

IANA should assign values indicated below.

TCP Header Flags

Bit Offset	Name	Reference	Assignment Notes
4	Reserved for future use	(this document)	
5	Reserved for future use	(this document)	
6	Reserved for future use	(this document)	
7	Reserved for future use	[RFC8311]	Previously used by Historic [RFC3540] as NS (Nonce Sum)
8	CWR (Congestion Window Reduced)	[RFC3168]	
9	ECE (ECN-Echo)	[RFC3168]	
10	Urgent Pointer field significant (URG)	(this document)	
11	Acknowledgment field significant (ACK)	(this document)	
12	Push Function (PSH)	(this document)	
13	Reset the connection (RST)	(this document)	
14	Synchronize sequence numbers (SYN)	(this document)	
15	No more data from sender (FIN)	(this document)	

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

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

6. Security and Privacy Considerations

The TCP design includes only rudimentary security features that improve the robustness and reliability of connections and application data transfer, but there are no built-in cryptographic capabilities to support any form of privacy, authentication, or other typical security functions. Non-cryptographic enhancements (e.g. [37]) have been developed to improve robustness of TCP connections to particular types of attacks, but the applicability and protections of non-cryptographic enhancements are limited (e.g. see section 1.1 of [37]). Applications typically utilize lower-layer (e.g. IPsec) and upper-layer (e.g. TLS) protocols to provide security and privacy for TCP connections and application data carried in TCP. Methods based on TCP options have been developed as well, to support some security capabilities.

In order to fully protect TCP connections (including their control flags) IPsec or the TCP Authentication Option (TCP-AO) [36] are the only current effective methods. Other methods discussed in this section may protect the payload, but either only a subset of the fields (e.g. tcpcrypt [59]) or none at all (e.g. TLS). Other

security features that have been added to TCP (e.g. ISN generation, sequence number checks, and others) are only capable of partially hindering attacks.

Applications using long-lived TCP flows have been vulnerable to attacks that exploit the processing of control flags described in earlier TCP specifications [30]. TCP-MD5 was a commonly implemented TCP option to support authentication for some of these connections, but had flaws and is now deprecated. TCP-AO provides a capability to protect long-lived TCP connections from attacks, and has superior properties to TCP-MD5. It does not provide any privacy for application data, nor for the TCP headers.

The "tcpcrypt" [59] Experimental extension to TCP provides the ability to cryptographically protect connection data. Metadata aspects of the TCP flow are still visible, but the application stream is well-protected. Within the TCP header, only the urgent pointer and FIN flag are protected through tcpcrypt.

The TCP Roadmap [48] includes notes about several RFCs related to TCP security. Many of the enhancements provided by these RFCs have been integrated into the present document, including ISN generation, mitigating blind in-window attacks, and improving handling of soft errors and ICMP packets. These are all discussed in greater detail in the referenced RFCs that originally described the changes needed to earlier TCP specifications. Additionally, see RFC 6093 [38] for discussion of security considerations related to the urgent pointer field, that has been deprecated.

Since TCP is often used for bulk transfer flows, some attacks are possible that abuse the TCP congestion control logic. An example is "ACK-division" attacks. Updates that have been made to the TCP congestion control specifications include mechanisms like Appropriate Byte Counting (ABC) [26] that act as mitigations to these attacks.

Other attacks are focused on exhausting the resources of a TCP server. Examples include SYN flooding [29] or wasting resources on non-progressing connections [40]. Operating systems commonly implement mitigations for these attacks. Some common defenses also utilize proxies, stateful firewalls, and other technologies outside of the end-host TCP implementation.

7. Acknowledgements

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

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

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

Michael Scharf
Yoshifumi Nishida
Pasi Sarolahti
Michael Tuexen

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

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

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

8. References

8.1. Normative References

[1] Postel, J., "Internet Protocol", STD 5, RFC 791, DOI 10.17487/RFC0791, September 1981, <https://www.rfc-editor.org/info/rfc791>.

[2] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, DOI 10.17487/RFC1191, November 1990, <https://www.rfc-editor.org/info/rfc1191>.

[3] McCann, J., Deering, S., and J. Mogul, "Path MTU Discovery for IP version 6", RFC 1981, DOI 10.17487/RFC1981, August 1996, <https://www.rfc-editor.org/info/rfc1981>.

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

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

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

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

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

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

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

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

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

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

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

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

8.2. Informative References

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

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

[18] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <https://www.rfc-editor.org/info/rfc1122>.

[19] Almquist, P., "Type of Service in the Internet Protocol Suite", RFC 1349, DOI 10.17487/RFC1349, July 1992, <https://www.rfc-editor.org/info/rfc1349>.

[20] Braden, R., "T/TCP -- TCP Extensions for Transactions Functional Specification", RFC 1644, DOI 10.17487/RFC1644, July 1994, <https://www.rfc-editor.org/info/rfc1644>.

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

[22] Paxson, V., Allman, M., Dawson, S., Fenner, W., Griner, J., Heavens, I., Lahey, K., Semke, J., and B. Volz, "Known TCP Implementation Problems", RFC 2525, DOI 10.17487/RFC2525, March 1999, <https://www.rfc-editor.org/info/rfc2525>.

[23] Xiao, X., Hannan, A., Paxson, V., and E. Crabbe, "TCP Processing of the IPv4 Precedence Field", RFC 2873, DOI 10.17487/RFC2873, June 2000, <https://www.rfc-editor.org/info/rfc2873>.

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

[25] Balakrishnan, H., Padmanabhan, V., Fairhurst, G., and M. Sooriyabandara, "TCP Performance Implications of Network Path Asymmetry", BCP 69, RFC 3449, DOI 10.17487/RFC3449, December 2002, <https://www.rfc-editor.org/info/rfc3449>.

[26] Allman, M., "TCP Congestion Control with Appropriate Byte Counting (ABC)", RFC 3465, DOI 10.17487/RFC3465, February 2003, <https://www.rfc-editor.org/info/rfc3465>.

[27] Fenner, B., "Experimental Values In IPv4, IPv6, ICMPv4, ICMPv6, UDP, and TCP Headers", RFC 4727, DOI 10.17487/RFC4727, November 2006, <https://www.rfc-editor.org/info/rfc4727>.

[28] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, DOI 10.17487/RFC4821, March 2007, <https://www.rfc-editor.org/info/rfc4821>.

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

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

[31] Culley, P., Elzur, U., Recio, R., Bailey, S., and J. Carrier, "Marker PDU Aligned Framing for TCP Specification", RFC 5044, DOI 10.17487/RFC5044, October 2007, <https://www.rfc-editor.org/info/rfc5044>.

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

[33] StJohns, M., Atkinson, R., and G. Thomas, "Common Architecture Label IPv6 Security Option (CALIPSO)", RFC 5570, DOI 10.17487/RFC5570, July 2009, <https://www.rfc-editor.org/info/rfc5570>.

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

[35] Sandlund, K., Pelletier, G., and L-E. Jonsson, "The RObust Header Compression (ROHC) Framework", RFC 5795, DOI 10.17487/RFC5795, March 2010, <https://www.rfc-editor.org/info/rfc5795>.

[36] 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>.

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

[38] Gont, F. and A. Yourtchenko, "On the Implementation of the TCP Urgent Mechanism", RFC 6093, DOI 10.17487/RFC6093, January 2011, <https://www.rfc-editor.org/info/rfc6093>.

[39] Gont, F., "Reducing the TIME-WAIT State Using TCP Timestamps", BCP 159, RFC 6191, DOI 10.17487/RFC6191, April 2011, <https://www.rfc-editor.org/info/rfc6191>.

[40] Bashyam, M., Jethanandani, M., and A. Ramaiah, "TCP Sender Clarification for Persist Condition", RFC 6429, DOI 10.17487/RFC6429, December 2011, <https://www.rfc-editor.org/info/rfc6429>.

[41] Gont, F. and S. Bellovin, "Defending against Sequence Number Attacks", RFC 6528, DOI 10.17487/RFC6528, February 2012, <https://www.rfc-editor.org/info/rfc6528>.

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

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

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

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

[46] Borman, D., Braden, B., Jacobson, V., and R. Scheffenegger, Ed., "TCP Extensions for High Performance", RFC 7323, DOI 10.17487/RFC7323, September 2014, <https://www.rfc-editor.org/info/rfc7323>.

[47] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <https://www.rfc-editor.org/info/rfc7413>.

[48] Duke, M., Braden, R., Eddy, W., Blanton, E., and A. Zimmermann, "A Roadmap for Transmission Control Protocol (TCP) Specification Documents", RFC 7414, DOI 10.17487/RFC7414, February 2015, <https://www.rfc-editor.org/info/rfc7414>.

[49] Black, D., Ed. and P. Jones, "Differentiated Services (Diffserv) and Real-Time Communication", RFC 7657, DOI 10.17487/RFC7657, November 2015, <https://www.rfc-editor.org/info/rfc7657>.

[50] Fairhurst, G. and M. Welzl, "The Benefits of Using Explicit Congestion Notification (ECN)", RFC 8087, DOI 10.17487/RFC8087, March 2017, <https://www.rfc-editor.org/info/rfc8087>.

[51] Fairhurst, G., Ed., Trammell, B., Ed., and M. Kuehlewind, Ed., "Services Provided by IETF Transport Protocols and Congestion Control Mechanisms", RFC 8095, DOI 10.17487/RFC8095, March 2017, <https://www.rfc-editor.org/info/rfc8095>.

[52] Welzl, M., Tuexen, M., and N. Khademi, "On the Usage of Transport Features Provided by IETF Transport Protocols", RFC 8303, DOI 10.17487/RFC8303, February 2018, <https://www.rfc-editor.org/info/rfc8303>.

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

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

[55] IANA, "Transmission Control Protocol (TCP) Parameters, https://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml", 2019.

[56] IANA, "Transmission Control Protocol (TCP) Header Flags, https://www.iana.org/assignments/tcp-header-flags/tcp-header-flags.xhtml", 2019.

[57] Gont, F., "Processing of IP Security/Compartment and Precedence Information by TCP", draft-gont-tcpm-tcp-seccomp-prec-00 (work in progress), March 2012.

[58] Gont, F. and D. Borman, "On the Validation of TCP Sequence Numbers", draft-gont-tcpm-tcp-seq-validation-02 (work in progress), March 2015.

[59] Bittau, A., Giffin, D., Handley, M., Mazieres, D., Slack, Q., and E. Smith, "Cryptographic protection of TCP Streams (tcpcrypt)", draft-ietf-tcpinc-tcpcrypt-09 (work in progress), November 2017.

[60] Touch, J. and W. Eddy, "TCP Extended Data Offset Option", draft-ietf-tcpm-tcp-edo-10 (work in progress), July 2018.

[61] Minshall, G., "A Proposed Modification to Nagle's Algorithm", draft-minshall-nagle-01 (work in progress), June 1999.

[62] Dalal, Y. and C. Sunshine, "Connection Management in Transport Protocols", Computer Networks Vol. 2, No. 6, pp. 454-473, December 1978.

[63] Faber, T., Touch, J., and W. Yui, "The TIME-WAIT state in TCP and Its Effect on Busy Servers", Proceedings of IEEE INFOCOM pp. 1573-1583, March 1999.

Appendix A. Other Implementation Notes

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

A.1. IP Security Compartment and Precedence

The IPv4 specification [1] includes a precedence value in the (now obsoleted) Type of Service field (TOS) field. It was modified in [19], and then obsoleted by the definition of Differentiated Services (DiffServ) [5]. Setting and conveying TOS between the network layer, TCP implementation, and applications is obsolete, and replaced by DiffServ in the current TCP specification.

RFC 793 requires checking the IP security compartment and precedence on incoming TCP segments for consistency within a connection, and with application requests. Each of these aspects of IP have become outdated, without specific updates to RFC 793. The issues with precedence were fixed by [23], which is Standards Track, and so this present TCP specification includes those changes. However, the state of IP security options that may be used by MLS systems is not as clean.

Resetting connections when incoming packets do not meet expected security compartment or precedence expectations has been recognized as a possible attack vector [57], and there has been discussion about amending the TCP specification to prevent connections from being aborted due to non-matching IP security compartment and DiffServ codepoint values.

A.1.1. Precedence

In DiffServ the former precedence values are treated as Class Selector codepoints, and methods for compatible treatment are described in the DiffServ architecture. The RFC 793/1122 TCP specification includes logic intending to have connections use the highest precedence requested by either endpoint application, and to keep the precedence consistent throughout a connection. This logic from the obsolete TOS is not applicable for DiffServ, and should not be included in TCP implementations, though changes to DiffServ values within a connection are discouraged. For discussion of this, see RFC 7657 (sec 5.1, 5.3, and 6) [49].

The obsoleted TOS processing rules in TCP assumed bidirectional (or symmetric) precedence values used on a connection, but the DiffServ architecture is asymmetric. Problems with the old TCP logic in this regard were described in [23] and the solution described is to ignore IP precedence in TCP. Since RFC 2873 is a Standards Track document (although not marked as updating RFC 793), current implementations are expected to be robust to these conditions. Note that the DiffServ field value used in each direction is a part of the interface between TCP and the network layer, and values in use can be indicated both ways between TCP and the application.

A.1.2. MLS Systems

The IP security option (IPSO) and compartment defined in [1] was refined in RFC 1038 that was later obsoleted by RFC 1108. The Commercial IP Security Option (CIPSO) is defined in FIPS-188, and is supported by some vendors and operating systems. RFC 1108 is now Historic, though RFC 791 itself has not been updated to remove the IP security option. For IPv6, a similar option (CALIPSO) has been

defined [33]. RFC 793 includes logic that includes the IP security/ compartment information in treatment of TCP segments. References to the IP "security/compartment" in this document may be relevant for Multi-Level Secure (MLS) system implementers, but can be ignored for non-MLS implementations, consistent with running code on the Internet. See Appendix A.1 for further discussion. Note that RFC 5570 describes some MLS networking scenarios where IPSO, CIPSO, or CALIPSO may be used. In these special cases, TCP implementers should see section 7.3.1 of RFC 5570, and follow the guidance in that document.

A.2. Sequence Number Validation

There are cases where the TCP sequence number validation rules can prevent ACK fields from being processed. This can result in connection issues, as described in [58], which includes descriptions of potential problems in conditions of simultaneous open, self-connects, simultaneous close, and simultaneous window probes. The document also describes potential changes to the TCP specification to mitigate the issue by expanding the acceptable sequence numbers.

In Internet usage of TCP, these conditions are rarely occurring. Common operating systems include different alternative mitigations, and the standard has not been updated yet to codify one of them, but implementers should consider the problems described in [58].

A.3. Nagle Modification

In common operating systems, both the Nagle algorithm and delayed acknowledgements are implemented and enabled by default. TCP is used by many applications that have a request-response style of communication, where the combination of the Nagle algorithm and delayed acknowledgements can result in poor application performance. A modification to the Nagle algorithm is described in [61] that improves the situation for these applications.

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

A.4. Low Water Mark Settings

Some operating system kernel TCP implementations include socket options that allow specifying the number of bytes in the buffer until

the socket layer will pass sent data to TCP (SO_SNDLOWAT) or to the application on receiving (SO_RCVLOWAT).

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

Appendix B. TCP Requirement Summary

This section is adapted from RFC 1122.

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

FEATURE	ReqID	MUST	SHOULD	MAY	SHOULD NOT	MUST NOT	Footnote
Push flag							
Aggregate or queue un-pushed data	MAY-16			x			
Sender collapse successive PSH flags	SHLD-27		x				
SEND call can specify PUSH	MAY-15			x			
If cannot: sender buffer indefinitely	MUST-60					x	
If cannot: PSH last segment	MUST-61	x					
Notify receiving ALP of PSH	MAY-17			x			1
Send max size segment when possible	SHLD-28		x				
Window							
Treat as unsigned number	MUST-1	x					
Handle as 32-bit number	REC-1		x				
Shrink window from right	SHLD-14				x		
- Send new data when window shrinks	SHLD-15				x		
- Retransmit old unacked data within window	SHLD-16		x				
- Time out conn for data past right edge	SHLD-17				x		
Robust against shrinking window	MUST-34	x					

Receiver's window closed indefinitely	MAY-8			x			
Use standard probing logic	MUST-35	x					
Sender probe zero window	MUST-36	x					
First probe after RTO	SHLD-29		x				
Exponential backoff	SHLD-30		x				
Allow window stay zero indefinitely	MUST-37	x					
Retransmit old data beyond SND.UNA+SND.WND	MAY-7			x			
Process RST and URG even with zero window	MUST-66	x					
Urgent Data							
Include support for urgent pointer	MUST-30	x					
Pointer indicates first non-urgent octet	MUST-62	x					
Arbitrary length urgent data sequence	MUST-31	x					
Inform ALP asynchronously of urgent data	MUST-32	x					1
ALP can learn if/how much urgent data Q'd	MUST-33	x					1
ALP employ the urgent mechanism	SHLD-13				x		
TCP Options							
Support the mandatory option set	MUST-4	x					
Receive TCP option in any segment	MUST-5	x					
Ignore unsupported options	MUST-6	x					
Include length for all options except EOL+NOP	MUST-68	x					
Cope with illegal option length	MUST-7	x					
Process options regardless of word alignment	MUST-64	x					
Implement sending & receiving MSS option	MUST-14	x					
IPv4 Send MSS option unless 536	SHLD-5		x				
IPv6 Send MSS option unless 1220	SHLD-5		x				
Send MSS option always	MAY-3			x			
IPv4 Send-MSS default is 536	MUST-15	x					
IPv6 Send-MSS default is 1220	MUST-15	x					
Calculate effective send seg size	MUST-16	x					
MSS accounts for varying MTU	SHLD-6		x				
MSS not sent on non-SYN segments	MUST-65					x	
MSS value based on MMS_R	MUST-67	x					
TCP Checksums							
Sender compute checksum	MUST-2	x					
Receiver check checksum	MUST-3	x					
ISN Selection							
Include a clock-driven ISN generator component	MUST-8	x					
Secure ISN generator with a PRF component	SHLD-1		x				
PRF computable from outside the host	MUST-9					x	
Opening Connections							
Support simultaneous open attempts	MUST-10	x					
SYN-RECEIVED remembers last state	MUST-11	x					
Passive Open call interfere with others	MUST-41					x	

Function: simultan. LISTENs for same port	MUST-42	x					
Ask IP for src address for SYN if necc.	MUST-44	x					
Otherwise, use local addr of conn.	MUST-45	x					
OPEN to broadcast/multicast IP Address	MUST-46					x	
Silently discard seg to bcast/mcast addr	MUST-57	x					
Closing Connections							
RST can contain data	SHLD-2		x				
Inform application of aborted conn	MUST-12	x					
Half-duplex close connections	MAY-1			x			
Send RST to indicate data lost	SHLD-3		x				
In TIME-WAIT state for 2MSL seconds	MUST-13	x					
Accept SYN from TIME-WAIT state	MAY-2			x			
Use Timestamps to reduce TIME-WAIT	SHLD-4		x				
Retransmissions							
Implement exponential backoff, slow start, and congestion avoidance	MUST-19	x					
Retransmit with same IP ident	MAY-4			x			
Karn's algorithm	MUST-18	x					
Generating ACKs:							
Aggregate whenever possible	MUST-58	x					
Queue out-of-order segments	SHLD-31		x				
Process all Q'd before send ACK	MUST-59	x					
Send ACK for out-of-order segment	MAY-13			x			
Delayed ACKs	SHLD-18		x				
Delay < 0.5 seconds	MUST-40	x					
Every 2nd full-sized segment ACK'd	SHLD-19	x					
Receiver SWS-Avoidance Algorithm	MUST-39	x					
Sending data							
Configurable TTL	MUST-49	x					
Sender SWS-Avoidance Algorithm	MUST-38	x					
Nagle algorithm	SHLD-7		x				
Application can disable Nagle algorithm	MUST-17	x					
Connection Failures:							
Negative advice to IP on R1 retxs	MUST-20	x					
Close connection on R2 retxs	MUST-20	x					
ALP can set R2	MUST-21	x					1
Inform ALP of R1<=retxs<R2	SHLD-9		x				1
Recommended value for R1	SHLD-10		x				
Recommended value for R2	SHLD-11		x				
Same mechanism for SYNs	MUST-22	x					
R2 at least 3 minutes for SYN	MUST-23	x					
Send Keep-alive Packets:	MAY-5			x			

- Application can request	MUST-24	x					
- Default is "off"	MUST-25	x					
- Only send if idle for interval	MUST-26	x					
- Interval configurable	MUST-27	x					
- Default at least 2 hrs.	MUST-28	x					
- Tolerant of lost ACKs	MUST-29	x					
- Send with no data	SHLD-12		x				
- Configurable to send garbage octet	MAY-6			x			
IP Options							
Ignore options TCP doesn't understand	MUST-50	x					
Time Stamp support	MAY-10			x			
Record Route support	MAY-11			x			
Source Route:							
ALP can specify	MUST-51	x					1
Overrides src rt in datagram	MUST-52	x					
Build return route from src rt	MUST-53	x					
Later src route overrides	SHLD-24		x				
Receiving ICMP Messages from IP	MUST-54	x					
Dest. Unreach (0,1,5) => inform ALP	SHLD-25		x				
Dest. Unreach (0,1,5) => abort conn	MUST-56					x	
Dest. Unreach (2-4) => abort conn	SHLD-26		x				
Source Quench => silent discard	MUST-55	x					
Time Exceeded => tell ALP, don't abort	MUST-56					x	
Param Problem => tell ALP, don't abort	MUST-56					x	
Address Validation							
Reject OPEN call to invalid IP address	MUST-46	x					
Reject SYN from invalid IP address	MUST-63	x					
Silently discard SYN to bcast/mcast addr	MUST-57	x					
TCP/ALP Interface Services							
Error Report mechanism	MUST-47	x					
ALP can disable Error Report Routine	SHLD-20		x				
ALP can specify DiffServ field for sending	MUST-48	x					
Passed unchanged to IP	SHLD-22		x				
ALP can change DiffServ field during connection	SHLD-21		x				
ALP generally changing DiffServ during conn.	SHLD-23				x		
Pass received DiffServ field up to ALP	MAY-9			x			
FLUSH call	MAY-14			x			
Optional local IP addr parm. in OPEN	MUST-43	x					
RFC 5961 Support:							
Implement data injection protection	MAY-12			x			
Explicit Congestion Notification:							
Support ECN	SHLD-8		x				

```
 ------------------------------------------------|--------|-|-|-|-|-|-
```

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

Author's Address

Wesley M. Eddy (editor)
MTI Systems
US

Email: wes@mti-systems.com

TCPM
Internet-Draft
Intended status: Standards Track
Expires: January 11, 2021

M. Scharf
Hochschule Esslingen
V. Murgai

M. Jethanandani
Kloud Services

July 10, 2020

YANG Model for Transmission Control Protocol (TCP) Configuration
draft-scharf-tcpm-yang-tcp-06

Abstract

This document specifies a YANG model for TCP on devices that are configured by network management protocols. The YANG model defines a container for all TCP connections and groupings of some of the parameters that can be imported and used in TCP implementations or by other models that need to configure TCP parameters. The model includes definitions from YANG Groupings for TCP Client and TCP Servers (I-D.ietf-netconf-tcp-client-server). The model is NMDA (RFC 8342) compliant.

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 January 11, 2021.

Copyright Notice

Table of Contents

1. Introduction

The Transmission Control Protocol (TCP) [RFC0793] is used by many applications in the Internet, including control and management protocols. Therefore, TCP is implemented on network elements that can be configured via network management protocols such as NETCONF [RFC6241] or RESTCONF [RFC8040]. This document specifies a YANG [RFC7950] 1.1 model for configuring TCP on network elements that support YANG data models, and is Network Management Datastore Architecture (NMDA) [RFC8342] compliant. This module defines a container for TCP connection, and includes definitions from YANG Groupings for TCP Clients and TCP Servers [I-D.ietf-netconf-tcp-client-server]. The model has a narrow scope and focuses on fundamental TCP functions and basic statistics. The model can be augmented or updated to address more advanced or implementation-specific TCP features in the future.

Many protocol stacks on Internet hosts use other methods to configure TCP, such as operating system configuration or policies. Many TCP/IP stacks cannot be configured by network management protocols such as NETCONF [RFC6241] or RESTCONF [RFC8040]. Moreover, many existing TCP/IP stacks do not use YANG data models. Such TCP implementations often have other means to configure the parameters listed in this document, which are outside the scope of this document.

This specification is orthogonal to the Management Information Base (MIB) for the Transmission Control Protocol (TCP) [RFC4022]. The basic statistics defined in this document follow the model of the TCP MIB. An TCP Extended Statistics MIB [RFC4898] is also available, but this document does not cover such extended statistics. It is possible also to translate a MIB into a YANG model, for instance using Translation of Structure of Management Information Version 2 (SMIv2) MIB Modules to YANG Modules [RFC6643]. However, this approach is not used in this document, as such a translated model would not be up-to-date.

There are other existing TCP-related YANG models, which are othogonal to this specification. Examples are:

- TCP header attributes are modeled in other models, such as YANG Data Model for Network Access Control Lists (ACLs) [RFC8519] and Distributed Denial-of-Service Open Thread Signaling (DOTS) Data Channel Specification [I-D.ietf-dots-data-channel].

- TCP-related configuration of a NAT (e.g., NAT44, NAT64, Destination NAT, ...) is defined in A YANG Module for Network Address Translation (NAT) and Network Prefix Translation (NPT) [RFC8512] and A YANG Data Model for Dual-Stack Lite (DS-Lite) [RFC8513].

2. Requirements Language

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

2.1. Note to RFC Editor

This document uses several placeholder values throughout the document. Please replace them as follows and remove this note before publication.

RFC XXXX, where XXXX is the number assigned to this document at the time of publication.

2020-07-10 with the actual date of the publication of this document.

3. Model Overview

3.1. Modeling Scope

TCP is implemented on many different system architectures. As a result, there are may different and often implementation-specific ways to configure parameters of the TCP protocol engine. In addition, in many TCP/IP stacks configuration exists for different scopes:

- Global configuration: Many TCP implementations have configuration parameters that affect all TCP connections. Typical examples include enabling or disabling optional protocol features.

- Interface configuration: It can be useful to use different TCP parameters on different interfaces, e.g., different device ports or IP interfaces. In that case, TCP parameters can be part of the interface configuration. Typical examples are the Maximum Segment Size (MSS) or configuration related to hardware offloading.

- Connection parameters: Many implementations have means to influence the behavior of each TCP connection, e.g., on the programming interface used by applications. A typical example are socket options in the socket API, such as disabling the Nagle algorithm by TCP_NODELAY. If an application uses such an interface, it is possible that the configuration of the application or application protocol includes TCP-related parameters. An example is the BGP YANG Model for Service Provider Networks [I-D.ietf-idr-bgp-model].

- Policies: Setting of TCP parameters can also be part of system policies, templates, or profiles. An example would be the preferences defined in An Abstract Application Layer Interface to Transport Services [I-D.ietf-taps-interface].

As a result, there is no ground truth for setting certain TCP parameters, and traditionally different TCP implementation have used different modeling approaches. For instance, one implementation may define a given configuration parameter globally, while another one uses per-interface settings, and both approaches work well for the corresponding use cases. Also, different systems may use different default values. In addition, TCP can be implemented in different

ways and design choices by the protocol engine often affect configuration options.

Nonetheless, a number of TCP stack parameters require configuration by YANG models. This document therefore defines a minimal YANG model with fundamental parameters directly following from TCP standards.

An important use case is the TCP configuration on network elements such as routers, which often use YANG data models. The model therefore specifies TCP parameters that are important on such TCP stacks. A typical example is the support of TCP-AO [RFC5925]. TCP-AO is increasingly supported on routers to secure routing protocols such as BGP. In that case, TCP-AO configuration is required on routers.

Given an installed base, the model also allows enabling of the legacy TCP MD5 [RFC2385] signature option. As the TCP MD5 signature option is obsoleted by TCP-AO, it is strongly RECOMMENDED to use TCP-AO.

Similar to the TCP MIB [RFC4022], this document also specifies basic statistics and a TCP connection table.

- Statistics: Counters for the number of active/passive opens, sent and received segments, errors, and possibly other detailed debugging information

- TCP connection table: Access to status information for all TCP connections

This allows implementations of TCP MIB [RFC4022] to migrate to the YANG model defined in this memo.

3.2. Model Design

The YANG model defined in this document includes definitions from the YANG Groupings for TCP Clients and TCP Servers [I-D.ietf-netconf-tcp-client-server]. Similar to that model, this specification defines YANG groupings. This allows reuse of these groupings in different YANG data models. It is intended that these groupings will be used either standalone or for TCP-based protocols as part of a stack of protocol-specific configuration models. An example could be the BGP YANG Model for Service Provider Networks [I-D.ietf-idr-bgp-model].

3.3. Tree Diagram

This section provides a abridged tree diagram for the YANG module defined in this document. Annotations used in the diagram are defined in YANG Tree Diagrams [RFC8340].

```
module: ietf-tcp
  +--rw tcp!
     +--rw connections
     |     ...
     +--rw server {server}?
     |     ...
     +--rw client {client}?
     |     ...
     +--ro statistics {statistics}?
           ...
```

4. TCP YANG Model

<CODE BEGINS> file "ietf-tcp@2020-07-10.yang"

```
module ietf-tcp {
  yang-version "1.1";
  namespace "urn:ietf:params:xml:ns:yang:ietf-tcp";
  prefix "tcp";

  import ietf-yang-types {
    prefix "yang";
    reference
      "RFC 6991: Common YANG Data Types.";
  }
  import ietf-tcp-client {
    prefix "tcpc";
  }
  import ietf-tcp-server {
    prefix "tcps";
  }
  import ietf-tcp-common {
    prefix "tcpcmn";
  }
  import ietf-inet-types {
    prefix "inet";
  }

  organization
    "IETF TCPM Working Group";

  contact
```

```
      "WG Web:   <http://tools.ietf.org/wg/tcpm>
       WG List:  <tcpm@ietf.org>

       Authors: Michael Scharf (michael.scharf at hs-esslingen dot de)
                Vishal Murgai (vmurgai at gmail dot com)
                Mahesh Jethanandani (mjethanandani at gmail dot com)";

    description
      "This module focuses on fundamental and standard TCP functions
       that widely implemented. The model can be augmented to address
       more advanced or implementation specific TCP features.";

    revision "2020-07-10" {
      description
        "Initial Version";
      reference
        "RFC XXX, TCP Configuration.";
    }

    // Features
    feature server {
      description
        "TCP Server configuration supported.";
    }

    feature client {
      description
        "TCP Client configuration supported.";
    }

    feature statistics {
      description
        "This implementation supports statistics reporting.";
    }

    // TCP-AO Groupings

    grouping ao {
      leaf enable-ao {
        type boolean;
        default "false";
        description
          "Enable support of TCP-Authentication Option (TCP-AO).";
      }

      leaf send-id {
        type uint8 {
          range "0..255";
```

```
        }
        must "../enable-ao = 'true'";
        description
          "The SendID is inserted as the KeyID of the TCP-AO option
           of outgoing segments.";
        reference
          "RFC 5925: The TCP Authentication Option.";
      }

      leaf recv-id {
        type uint8 {
          range "0..255";
        }
        must "../enable-ao = 'true'";
        description
          "The RecvID is matched against the TCP-AO KeyID of incoming
           segments.";
        reference
          "RFC 5925: The TCP Authentication Option.";
      }

      leaf include-tcp-options {
        type boolean;
        must "../enable-ao = 'true'";
        description
          "Include TCP options in HMAC calculation.";
      }

      leaf accept-ao-mismatch {
        type boolean;
        must "../enable-ao = 'true'";
        description
          "Accept packets with HMAC mismatch.";
      }
      description
        "Authentication Option (AO) for TCP.";
      reference
        "RFC 5925: The TCP Authentication Option.";
    }

    // MD5 grouping

    grouping md5 {
      description
        "Grouping for use in authenticating TCP sessions using MD5.";
      reference
        "RFC 2385: Protection of BGP Sessions via the TCP MD5
         Signature.";
```

```
      leaf enable-md5 {
        type boolean;
        default "false";
        description
          "Enable support of MD5 to authenticate a TCP session.";
      }
    }

    // TCP configuration

    container tcp {
      presence "The container for TCP configuration.";

      description
        "TCP container.";

      container connections {
        list connection {
          key "local-address remote-address local-port remote-port";

          leaf local-address {
            type inet:ip-address;
            description
              "Local address that forms the connection identifier.";
          }

          leaf remote-address {
            type inet:ip-address;
            description
              "Remote address that forms the connection identifier.";
          }

          leaf local-port {
            type inet:port-number;
            description
              "Local TCP port that forms the connection identifier.";
          }

          leaf remote-port {
            type inet:port-number;
            description
              "Remote TCP port that forms the connection identifier.";
          }

          container common {
            uses tcpcmn:tcp-common-grouping;

            choice authentication {
```

```
            case ao {
              uses ao;
              description
                "Use TCP-AO to secure the connection.";
            }

            case md5 {
              uses md5;
              description
                "Use TCP-MD5 to secure the connection.";
            }
            description
              "Choice of how to secure the TCP connection.";
          }
          description
            "Common definitions of TCP configuration. This includes
             parameters such as how to secure the connection,
             that can be part of either the client or server.";
        }
        description
          "Connection related parameters.";
      }
      description
        "A container of all TCP connections.";
    }

    container server {
      if-feature server;
      uses tcps:tcp-server-grouping;
      description
        "Definitions of TCP server configuration.";
    }

    container client {
      if-feature client;
      uses tcpc:tcp-client-grouping;
      description
        "Definitions of TCP client configuration.";
    }

    container statistics {
      if-feature statistics;
      config false;

      leaf active-opens {
        type yang:counter32;
        description
          "The number of times that TCP connections have made a direct
```

```
           transition to the SYN-SENT state from the CLOSED state.";
        }

        leaf passive-opens {
          type yang:counter32;
          description
            "The number of times TCP connections have made a direct
             transition to the SYN-RCVD state from the LISTEN state.";
        }

        leaf attempt-fails {
          type yang:counter32;
          description
            "The number of times that TCP connections have made a direct
             transition to the CLOSED state from either the SYN-SENT
             state or the SYN-RCVD state, plus the number of times that
             TCP connections have made a direct transition to the
             LISTEN state from the SYN-RCVD state.";
        }

        leaf establish-resets {
          type yang:counter32;
          description
            "The number of times that TCP connections have made a direct
             transition to the CLOSED state from either the ESTABLISHED
             state or the CLOSE-WAIT state.";
        }

        leaf currently-established {
          type yang:gauge32;
          description
            "The number of TCP connections for which the current state
             is either ESTABLISHED or CLOSE-WAIT.";
        }

        leaf in-segments {
          type yang:counter64;
          description
            "The total number of segments received, including those
             received in error.  This count includes segments received
             on currently established connections.";
        }

        leaf out-segments {
          type yang:counter64;
          description
            "The total number of segments sent, including those on
             current connections but excluding those containing only
```

```
            retransmitted octets.";
        }

        leaf retransmitted-segments {
          type yang:counter32;
          description
            "The total number of segments retransmitted; that is, the
             number of TCP segments transmitted containing one or more
             previously transmitted octets.";
        }

        leaf in-errors {
          type yang:counter32;
          description
            "The total number of segments received in error (e.g., bad
             TCP checksums).";
        }

        leaf out-resets {
          type yang:counter32;
          description
            "The number of TCP segments sent containing the RST flag.";
        }

        action reset {
          description
            "Reset statistics action command.";
          input {
            leaf reset-at {
              type yang:date-and-time;
              description
                "Time when the reset action needs to be
                 executed.";
            }
          }
          output {
            leaf reset-finished-at {
              type yang:date-and-time;
              description
                "Time when the reset action command completed.";
            }
          }
        }
        description
          "Statistics across all connections.";
      }
    }
  }
```

<CODE ENDS>

5. IANA Considerations

5.1. The IETF XML Registry

This document registers two URIs in the "ns" subregistry of the IETF XML Registry [RFC3688]. Following the format in IETF XML Registry [RFC3688], the following registrations are requested:

URI: urn:ietf:params:xml:ns:yang:ietf-tcp
Registrant Contact: The TCPM WG of the IETF.
XML: N/A, the requested URI is an XML namespace.

5.2. The YANG Module Names Registry

This document registers a YANG modules in the YANG Module Names registry YANG - A Data Modeling Language [RFC6020]. Following the format in YANG - A Data Modeling Language [RFC6020], the following registrations are requested:

```
name:         ietf-tcp
namespace:    urn:ietf:params:xml:ns:yang:ietf-tcp
prefix:       tcp
reference:    RFC XXXX
```

6. Security Considerations

The YANG module specified in this document defines a schema for data that is designed to be accessed via network management protocols such as NETCONF [RFC6241] or RESTCONF [RFC8040]. The lowest NETCONF layer is the secure transport layer, and the mandatory-to-implement secure transport is Secure Shell (SSH) described in Using the NETCONF protocol over SSH [RFC6242]. The lowest RESTCONF layer is HTTPS, and the mandatory-to-implement secure transport is TLS [RFC8446].

The Network Configuration Access Control Model (NACM) [RFC8341] provides the means to restrict access for particular NETCONF or RESTCONF users to a preconfigured subset of all available NETCONF or RESTCONF protocol operations and content.

7. References

7.1. Normative References

[I-D.ietf-netconf-tcp-client-server]
Watsen, K. and M. Scharf, "YANG Groupings for TCP Clients and TCP Servers", draft-ietf-netconf-tcp-client-server-06 (work in progress), June 2020.

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

[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>.

[RFC2385] Heffernan, A., "Protection of BGP Sessions via the TCP MD5 Signature Option", RFC 2385, DOI 10.17487/RFC2385, August 1998, <https://www.rfc-editor.org/info/rfc2385>.

[RFC3688] Mealling, M., "The IETF XML Registry", BCP 81, RFC 3688, DOI 10.17487/RFC3688, January 2004, <https://www.rfc-editor.org/info/rfc3688>.

[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>.

[RFC6020] Bjorklund, M., Ed., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020, DOI 10.17487/RFC6020, October 2010, <https://www.rfc-editor.org/info/rfc6020>.

[RFC6241] Enns, R., Ed., Bjorklund, M., Ed., Schoenwaelder, J., Ed., and A. Bierman, Ed., "Network Configuration Protocol (NETCONF)", RFC 6241, DOI 10.17487/RFC6241, June 2011, <https://www.rfc-editor.org/info/rfc6241>.

[RFC6242] Wasserman, M., "Using the NETCONF Protocol over Secure Shell (SSH)", RFC 6242, DOI 10.17487/RFC6242, June 2011, <https://www.rfc-editor.org/info/rfc6242>.

[RFC7950] Bjorklund, M., Ed., "The YANG 1.1 Data Modeling Language", RFC 7950, DOI 10.17487/RFC7950, August 2016, <https://www.rfc-editor.org/info/rfc7950>.

[RFC8040] Bierman, A., Bjorklund, M., and K. Watsen, "RESTCONF Protocol", RFC 8040, DOI 10.17487/RFC8040, January 2017, <https://www.rfc-editor.org/info/rfc8040>.

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

[RFC8340] Bjorklund, M. and L. Berger, Ed., "YANG Tree Diagrams", BCP 215, RFC 8340, DOI 10.17487/RFC8340, March 2018, <https://www.rfc-editor.org/info/rfc8340>.

[RFC8341] Bierman, A. and M. Bjorklund, "Network Configuration Access Control Model", STD 91, RFC 8341, DOI 10.17487/RFC8341, March 2018, <https://www.rfc-editor.org/info/rfc8341>.

[RFC8342] Bjorklund, M., Schoenwaelder, J., Shafer, P., Watsen, K., and R. Wilton, "Network Management Datastore Architecture (NMDA)", RFC 8342, DOI 10.17487/RFC8342, March 2018, <https://www.rfc-editor.org/info/rfc8342>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

7.2. Informative References

[I-D.ietf-dots-data-channel] Boucadair, M. and T. Reddy.K, "Distributed Denial-of-Service Open Threat Signaling (DOTS) Data Channel Specification", draft-ietf-dots-data-channel-31 (work in progress), July 2019.

[I-D.ietf-idr-bgp-model] Jethanandani, M., Patel, K., Hares, S., and J. Haas, "BGP YANG Model for Service Provider Networks", draft-ietf-idr-bgp-model-09 (work in progress), June 2020.

[I-D.ietf-taps-interface] Trammell, B., Welzl, M., Enghardt, T., Fairhurst, G., Kuehlewind, M., Perkins, C., Tiesel, P., Wood, C., and T. Pauly, "An Abstract Application Layer Interface to Transport Services", draft-ietf-taps-interface-06 (work in progress), March 2020.

[RFC4022] Raghunarayan, R., Ed., "Management Information Base for the Transmission Control Protocol (TCP)", RFC 4022, DOI 10.17487/RFC4022, March 2005, <https://www.rfc-editor.org/info/rfc4022>.

[RFC4898] Mathis, M., Heffner, J., and R. Raghunarayan, "TCP Extended Statistics MIB", RFC 4898, DOI 10.17487/RFC4898, May 2007, <https://www.rfc-editor.org/info/rfc4898>.

[RFC6643] Schoenwaelder, J., "Translation of Structure of Management Information Version 2 (SMIv2) MIB Modules to YANG Modules", RFC 6643, DOI 10.17487/RFC6643, July 2012, <https://www.rfc-editor.org/info/rfc6643>.

[RFC8512] Boucadair, M., Ed., Sivakumar, S., Jacquenet, C., Vinapamula, S., and Q. Wu, "A YANG Module for Network Address Translation (NAT) and Network Prefix Translation (NPT)", RFC 8512, DOI 10.17487/RFC8512, January 2019, <https://www.rfc-editor.org/info/rfc8512>.

[RFC8513] Boucadair, M., Jacquenet, C., and S. Sivakumar, "A YANG Data Model for Dual-Stack Lite (DS-Lite)", RFC 8513, DOI 10.17487/RFC8513, January 2019, <https://www.rfc-editor.org/info/rfc8513>.

[RFC8519] Jethanandani, M., Agarwal, S., Huang, L., and D. Blair, "YANG Data Model for Network Access Control Lists (ACLs)", RFC 8519, DOI 10.17487/RFC8519, March 2019, <https://www.rfc-editor.org/info/rfc8519>.

Appendix A. Acknowledgements

Michael Scharf was supported by the StandICT.eu project, which is funded by the European Commission under the Horizon 2020 Programme.

The following persons have contributed to this document by reviews: Mohamed Boucadair

Appendix B. Changes compared to previous versions

Changes compared to draft-scharf-tcpm-yang-tcp-04

- Removed congestion control
- Removed global stack parameters

Changes compared to draft-scharf-tcpm-yang-tcp-03

- Updated TCP-AO grouping
- Added congestion control

Changes compared to draft-scharf-tcpm-yang-tcp-02

- Initial proposal of a YANG model including base configuration parameters, TCP-AO configuration, and a connection list
- Editorial bugfixes and outdated references reported by Mohamed Boucadair
- Additional co-author Mahesh Jethanandani

Changes compared to draft-scharf-tcpm-yang-tcp-01

- Alignment with [I-D.ietf-netconf-tcp-client-server]
- Removing backward-compatibility to the TCP MIB
- Additional co-author Vishal Murgai

Changes compared to draft-scharf-tcpm-yang-tcp-00

- Editorial improvements

Appendix C. Examples

C.1. Keepalive Configuration

This particular example demonstrates how both a particular connection can be configured for keepalives.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
 This example shows how TCP keepalive can be configured for
 a given connection. An idle connection is dropped after
 idle-time + (max-probes * probe-interval).
-->
<config xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <tcp
      xmlns="urn:ietf:params:xml:ns:yang:ietf-tcp">
    <connections>
      <connection>
        <local-address>192.168.1.1</local-address>
        <remote-address>192.168.1.2</remote-address>
        <local-port>1025</local-port>
        <remote-port>80</remote-port>
        <common>
          <keepalives>
            <idle-time>5</idle-time>
            <max-probes>5</max-probes>
            <probe-interval>10</probe-interval>
          </keepalives>
        </common>
      </connection>
    </connections>
    <!--
     It is not clear why a server and client configuration is
     needed here even as they under a feature statement and therefore
     are required only if the feature is declared. Adding it so
     that yanglint allows this validation to run.
    -->
    <server>
      <local-address>192.168.1.1</local-address>
    </server>
    <client>
      <remote-address>192.168.1.2</remote-address>
    </client>
  </tcp>
</config>
```

Authors' Addresses

Michael Scharf
Hochschule Esslingen - University of Applied Sciences
Flandernstr. 101
Esslingen 73732
Germany

Email: michael.scharf@hs-esslingen.de

Vishal Murgai

Email: vmurgai@gmail.com

Mahesh Jethanandani
Kloud Services

Email: mjethanandani@gmail.com

TCPM
Internet Draft
Intended status: Informational
Expires: June 2021

J. Touch
Independent consultant
J. Kuusisaari
Infinera

December 23, 2020

TCP-AO Test Vectors

draft-touch-tcpm-ao-test-vectors-02.txt

Status of this Memo

Copyright Notice

Abstract

This document provides test vectors to validate implementations of the two mandatory authentication algorithms specified for the TCP Authentication Option over both IPv4 and IPv6. This includes validation of the key derivation function (KDF) based on a set of test connection parameters as well as validation of the message authentication code (MAC). Vectors are provided for both currently required pairs of KDF and MAC algorithms: one based on SHA-1 and the other on AES-128. The vectors also validate both whole TCP segments as well as segments whose options are excluded for NAT traversal.

Table of Contents

1. Introduction

This document provides test vectors to validate the correct implementation of the TCP Authentication Option (TCP-AO) [RFC5925]. It includes the specification of all endpoint parameters to generate the variety of TCP segments covered by different keys and MAC coverage, i.e., both the default case and the variant where TCP

options are ignored. It also includes both default key derivation functions (KDFs) and MAC generation algorithms [RFC5926].

The experimental extension to support NAT traversal is not included in the provided test vectors [RFC6978].

This document provides test vectors from an implementation.

2. Conventions used in this document

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

3. Input Test Vectors

3.1. TCP Connection Parameters

The following parameters are used throughout this suite of test vectors. The terms ’active’ and ’passive’ are used as defined for TCP [RFC793].

3.1.1. TCP-AO parameters

The following values are used for all exchanges. This suite does not test key switchover. The KeyIDs are as indicated for TCP-AO [RFC5925]. The Master Key is used to derive the traffic keys [RFC5926].

Active (client) side KeyID: 61 (3D)

Passive (server) side KeyID: 84 (54)

Master_key: "testvector" (length = 10 bytes)

3.1.2. Active (client) side parameters

The following endpoint parameters are used on the active side of the TCP connection, i.e., the side that initiates the TCP SYN.

For IPv4: 10.11.12.13

For IPv6: FD00::1

TCP port: (varies)

3.1.3. Passive (server) side parameters

The following endpoint parameters are used for the passive side of the TCP connection, i.e., the side that responds with a TCP SYN-ACK.

For IPv4: 172.27.28.29

For IPv6: FD00::2

TCP port = 179 (BGP)

3.1.4. Other IP fields and options

No IP options are used in these test vectors.

All IPv4 packets use the following other parameters [RFC791]: DSCP = 111000 (CS7) as is typical for BGP, ECN = 00, set DF, and clear MF.

IPv4 uses a TTL of 255; IPv6 uses a hopcount of 64.

All IPv6 use the following other parameters [RFC8200]: (TBD).

3.1.5. Other TCP fields and options

The SYN and SYN-ACK segments include MSS [RFC793], NOP, WindowScale [RFC7323], SACK Permitted [RFC2018], TimeStamp [RFC7323], and TCP-AO [RFC5925], in that order.

All other example segments include NOP, NOP, TimeStamp, and TCP-AO, in that order.

All segment URG pointers are zero [RFC793]. All segments with data set the PSH flag [RFC793].

4. IPv4 SHA-1 Output Test Vectors

SHA-1 is computed as specified for TCP-AO [RFC5926].

4.1. SHA-1 MAC (default - covers TCP options)

4.1.1. Send (client) SYN (covers options)

Send_SYN_traffic_key:

```
6d 63 ef 1b 02 fe 15 09 d4 b1 40 27 07 fd 7b 04
16 ab b7 4f
```

IPv4/TCP:

```
45 e0 00 4c dd 0f 40 00 ff 06 bf 6b 0a 0b 0c 0d
ac 1b 1c 1d e9 d7 00 b3 fb fb ab 5a 00 00 00 00
e0 02 ff ff ca c4 00 00 02 04 05 b4 01 03 03 08
04 02 08 0a 00 15 5a b7 00 00 00 00 1d 10 3d 54
2e e4 37 c6 f8 ed e6 d7 c4 d6 02 e7
```

MAC:

```
2e e4 37 c6 f8 ed e6 d7 c4 d6 02 e7
```

4.1.2. Receive (server) SYN-ACK (covers options)

Receive_SYN_traffic_key:

```
d9 e2 17 e4 83 4a 80 ca 2f 3f d8 de 2e 41 b8 e6
79 7f ea 96
```

IPv4/TCP:

```
45 e0 00 4c 65 06 40 00 ff 06 37 75 ac 1b 1c 1d
0a 0b 0c 0d 00 b3 e9 d7 11 c1 42 61 fb fb ab 5b
e0 12 ff ff 37 76 00 00 02 04 05 b4 01 03 03 08
04 02 08 0a 84 a5 0b eb 00 15 5a b7 1d 10 54 3d
ee ab 0f e2 4c 30 10 81 51 16 b3 be
```

MAC:

```
ee ab 0f e2 4c 30 10 81 51 16 b3 be
```

4.1.3. Send (client) non-SYN (covers options)

Send_other_traffic_key:

```
d2 e5 9c 65 ff c7 b1 a3 93 47 65 64 63 b7 0e dc
24 a1 3d 71
```

IPv4/TCP:

```
45 e0 00 87 36 a1 40 00 ff 06 65 9f 0a 0b 0c 0d
ac 1b 1c 1d e9 d7 00 b3 fb fb ab 5b 11 c1 42 62
c0 18 01 04 a1 62 00 00 01 01 08 0a 00 15 5a c1
84 a5 0b eb 1d 10 3d 54 70 64 cf 99 8c c6 c3 15
c2 c2 e2 bf ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff 00 43 01 04 da bf 00 b4 0a 0b 0c 0d
26 02 06 01 04 00 01 00 01 02 02 80 00 02 02 02
00 02 02 42 00 02 06 41 04 00 00 da bf 02 08 40
06 00 64 00 01 01 00
```

MAC:

```
70 64 cf 99 8c c6 c3 15 c2 c2 e2 bf
```

4.1.4. Receive (server) non-SYN (covers options)

Receive_other_traffic_key:

```
d9 e2 17 e4 83 4a 80 ca 2f 3f d8 de 2e 41 b8 e6
79 7f ea 96
```

IPv4/TCP:

```
45 e0 00 87 1f a9 40 00 ff 06 7c 97 ac 1b 1c 1d
0a 0b 0c 0d 00 b3 e9 d7 11 c1 42 62 fb fb ab 9e
c0 18 01 00 40 0c 00 00 01 01 08 0a 84 a5 0b f5
00 15 5a c1 1d 10 54 3d a6 3f 0e cb bb 2e 63 5c
95 4d ea c7 ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff 00 43 01 04 da c0 00 b4 ac 1b 1c 1d
26 02 06 01 04 00 01 00 01 02 02 80 00 02 02 02
00 02 02 42 00 02 06 41 04 00 00 da c0 02 08 40
06 00 64 00 01 01 00
```

MAC:

```
a6 3f 0e cb bb 2e 63 5c 95 4d ea c7
```

4.2. SHA-1 MAC (omits TCP options)

4.2.1. Send (client) SYN (omits options)

Send_SYN_traffic_key:

```
30 ea a1 56 0c f0 be 57 da b5 c0 45 22 9f b1 0a
42 3c d7 ea
```

IPv4/TCP:

```
45 e0 00 4c 53 99 40 00 ff 06 48 e2 0a 0b 0c 0d
ac 1b 1c 1d ff 12 00 b3 cb 0e fb ee 00 00 00 00
e0 02 ff ff 54 1f 00 00 02 04 05 b4 01 03 03 08
04 02 08 0a 00 02 4c ce 00 00 00 00 1d 10 3d 54
80 af 3c fe b8 53 68 93 7b 8f 9e c2
```

MAC:

```
80 af 3c fe b8 53 68 93 7b 8f 9e c2
```

4.2.2. Receive (server) SYN-ACK (omits options)

Receive_SYN_traffic_key:

```
b5 b2 89 6b b3 66 4e 81 76 b0 ed c6 e7 99 52 41a
01 a8 30 7f
```

IPv4/TCP:

```
45 e0 00 4c 32 84 40 00 ff 06 69 f7 ac 1b 1c 1d
0a 0b 0c 0d 00 b3 ff 12 ac d5 b5 e1 cb 0e fb ef
e0 12 ff ff 38 8e 00 00 02 04 05 b4 01 03 03 08
04 02 08 0a 57 67 72 f3 00 02 4c ce 1d 10 54 3d
09 30 6f 9a ce a6 3a 8c 68 cb 9a 70
```

MAC:

```
09 30 6f 9a ce a6 3a 8c 68 cb 9a 70
```

4.2.3. Send (client) non-SYN (omits options)

Send_other_traffic_key:

```
f3 db 17 93 d7 91 0e cd 80 6c 34 f1 55 ea 1f 00
34 59 53 e3
```

IPv4/TCP:

```
45 e0 00 87 a8 f5 40 00 ff 06 f3 4a 0a 0b 0c 0d
ac 1b 1c 1d ff 12 00 b3 cb 0e fb ef ac d5 b5 e2
c0 18 01 04 6c 45 00 00 01 01 08 0a 00 02 4c ce
57 67 72 f3 1d 10 3d 54 71 06 08 cc 69 6c 03 a2
71 c9 3a a5 ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff 00 43 01 04 da bf 00 b4 0a 0b 0c 0d
26 02 06 01 04 00 01 00 01 02 02 80 00 02 02 02
00 02 02 42 00 02 06 41 04 00 00 da bf 02 08 40
06 00 64 00 01 01 00
```

MAC:

```
71 06 08 cc 69 6c 03 a2 71 c9 3a a5
```

4.2.4. Receive (server) non-SYN (omits options)

Receive_other_traffic_key:

```
b5 b2 89 6b b3 66 4e 81 76 b0 ed c6 e7 99 52 41
01 a8 30 7f
```

IPv4/TCP:

```
45 e0 00 87 54 37 40 00 ff 06 48 09 ac 1b 1c 1d
0a 0b 0c 0d 00 b3 ff 12 ac d5 b5 e2 cb 0e fc 32
c0 18 01 00 46 b6 00 00 01 01 08 0a 57 67 72 f3
00 02 4c ce 1d 10 54 3d 97 76 6e 48 ac 26 2d e9
ae 61 b4 f9 ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff 00 43 01 04 da c0 00 b4 ac 1b 1c 1d
26 02 06 01 04 00 01 00 01 02 02 80 00 02 02 02
00 02 02 42 00 02 06 41 04 00 00 da c0 02 08 40
06 00 64 00 01 01 00
```

MAC:

```
97 76 6e 48 ac 26 2d e9 ae 61 b4 f9
```

5. IPv4 AES-128 Output Test Vectors

AES-128 is computed as required by TCP-AO [RFC5926].

5.1. AES MAC (default - covers TCP options)

5.1.1. Send (client) SYN (covers options)

Send_SYN_traffic_key:

```
f5 b8 b3 d5 f3 4f db b6 eb 8d 4a b9 66 0e 60 e3
```

IP/TCP:

```
45 e0 00 4c 7b 9f 40 00 ff 06 20 dc 0a 0b 0c 0d
ac 1b 1c 1d c4 fa 00 b3 78 7a 1d df 00 00 00 00
e0 02 ff ff 5a 0f 00 00 02 04 05 b4 01 03 03 08
04 02 08 0a 00 01 7e d0 00 00 00 00 1d 10 3d 54
e4 77 e9 9c 80 40 76 54 98 e5 50 91
```

MAC:

```
e4 77 e9 9c 80 40 76 54 98 e5 50 91
```

5.1.2. Receive (server) SYN-ACK (covers options)

Receive_SYN_traffic_key:

```
4b c7 57 1a 48 6f 32 64 bb d8 88 47 40 66 b4 b1
```

IPv4/TCP:

```
45 e0 00 4c 4b ad 40 00 ff 06 50 ce ac 1b 1c 1d
0a 0b 0c 0d 00 b3 c4 fa fa dd 6d e9 78 7a 1d e0
e0 12 ff ff f3 f2 00 00 02 04 05 b4 01 03 03 08
04 02 08 0a 93 f4 e9 e8 00 01 7e d0 1d 10 54 3d
d6 ad a7 bc 4c dd 53 6d 17 69 db 5f
```

MAC:

```
d6 ad a7 bc 4c dd 53 6d 17 69 db 5f
```

5.1.3. Send (client) non-SYN (covers options)

Send_other_traffic_key:

```
8c 8a e0 e8 37 1e c5 cb b9 7e a7 9d 90 41 83 91
```

IPv4/TCP:

```
45 e0 00 87 fb 4f 40 00 ff 06 a0 f0 0a 0b 0c 0d
ac 1b 1c 1d c4 fa 00 b3 78 7a 1d e0 fa dd 6d ea
c0 18 01 04 95 05 00 00 01 01 08 0a 00 01 7e d0
93 f4 e9 e8 1d 10 3d 54 77 41 27 42 fa 4d c4 33
ef f0 97 3e ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff 00 43 01 04 da bf 00 b4 0a 0b 0c 0d
26 02 06 01 04 00 01 00 01 02 02 80 00 02 02 02
00 02 02 42 00 02 06 41 04 00 00 da bf 02 08 40
06 00 64 00 01 01 00
```

MAC:

```
77 41 27 42 fa 4d c4 33 ef f0 97 3e
```

5.1.4. Receive (server) non-SYN (covers options)

Receive_other_traffic_key:

```
4b c7 57 1a 48 6f 32 64 bb d8 88 47 40 66 b4 b1
```

IPv4/TCP:

```
45 e0 00 87 b9 14 40 00 ff 06 e3 2b ac 1b 1c 1d
0a 0b 0c 0d 00 b3 c4 fa fa dd 6d ea 78 7a 1e 23
c0 18 01 00 e7 db 00 00 01 01 08 0a 93 f4 e9 e8
00 01 7e d0 1d 10 54 3d f6 d9 65 a7 83 82 a7 48
45 f7 2d ac ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff 00 43 01 04 da c0 00 b4 ac 1b 1c 1d
26 02 06 01 04 00 01 00 01 02 02 80 00 02 02 02
00 02 02 42 00 02 06 41 04 00 00 da c0 02 08 40
06 00 64 00 01 01 00
```

MAC:

```
f6 d9 65 a7 83 82 a7 48 45 f7 2d ac
```

5.2. AES MAC (omits TCP options)

5.2.1. Send (client) SYN (omits options)

Send_SYN_traffic_key:

```
2c db ae 13 92 c4 94 49 fa 92 c4 50 97 35 d5 0e
```

IPv4/TCP:

```
45 e0 00 4c f2 2e 40 00 ff 06 aa 4c 0a 0b 0c 0d
ac 1b 1c 1d da 1c 00 b3 38 9b ed 71 00 00 00 00
e0 02 ff ff 70 bf 00 00 02 04 05 b4 01 03 03 08
04 02 08 0a 00 01 85 e1 00 00 00 00 1d 10 3d 54
c4 4e 60 cb 31 f7 c0 b1 de 3d 27 49
```

MAC:

```
c4 4e 60 cb 31 f7 c0 b1 de 3d 27 49
```

5.2.2. Receive (server) SYN-ACK (omits options)

Receive_SYN_traffic_key:

```
3c e6 7a 55 18 69 50 6b 63 47 b6 33 c5 0a 62 4a
```

IPv4/TCP:

```
45 e0 00 4c 6c c0 40 00 ff 06 2f bb ac 1b 1c 1d
0a 0b 0c 0d 00 b3 da 1c d3 84 4a 6f 38 9b ed 72
e0 12 ff ff e4 45 00 00 02 04 05 b4 01 03 03 08
04 02 08 0a ce 45 98 38 00 01 85 e1 1d 10 54 3d
3a 6a bb 20 7e 49 b1 be 71 36 db 90
```

MAC:

```
3a 6a bb 20 7e 49 b1 be 71 36 db 90
```

5.2.3. Send (client) non-SYN (omits options)

Send_other_traffic_key:

```
03 5b c4 00 a3 41 ff e5 95 f5 9f 58 00 50 06 ca
```

IPv4/TCP:

```
45 e0 00 87 ee 91 40 00 ff 06 ad ae 0a 0b 0c 0d
ac 1b 1c 1d da 1c 00 b3 38 9b ed 72 d3 84 4a 70
c0 18 01 04 88 51 00 00 01 01 08 0a 00 01 85 e1
ce 45 98 38 1d 10 3d 54 75 85 e9 e9 d5 c3 ec 85
7b 96 f8 37 ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff 00 43 01 04 da bf 00 b4 0a 0b 0c 0d
26 02 06 01 04 00 01 00 01 02 02 80 00 02 02 02
00 02 02 42 00 02 06 41 04 00 00 da bf 02 08 40
06 00 64 00 01 01 00
```

MAC:

```
75 85 e9 e9 d5 c3 ec 85 7b 96 f8 37
```

5.2.4. Receive (server) non-SYN (omits options)

Receive_other_traffic_key:

```
3c e6 7a 55 18 69 50 6b 63 47 b6 33 c5 0a 62 4a
```

IPv4/TCP:

```
45 e0 00 87 6a 21 40 00 ff 06 32 1f ac 1b 1c 1d
0a 0b 0c 0d 00 b3 da 1c d3 84 4a 70 38 9b ed 72
c0 18 01 00 04 49 00 00 01 01 08 0a ce 45 98 38
00 01 85 e1 1d 10 54 3d 5c 04 0f d9 23 33 04 76
5c 09 82 f4 ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff 00 43 01 04 da c0 00 b4 ac 1b 1c 1d
26 02 06 01 04 00 01 00 01 02 02 80 00 02 02 02
00 02 02 42 00 02 06 41 04 00 00 da c0 02 08 40
06 00 64 00 01 01 00
```

MAC:

```
5c 04 0f d9 23 33 04 76 5c 09 82 f4
```

6. IPv6 SHA-1 Output Test Vectors

SHA-1 is computed as specified for TCP-AO [RFC5926].

6.1. SHA-1 MAC (default - covers TCP options)

6.1.1. Send (client) SYN (covers options)

Send_SYN_traffic_key:

```
62 5e c0 9d 57 58 36 ed c9 b6 42 84 18 bb f0 69
89 a3 61 bb
```

IPv6/TCP:

```
6e 08 91 dc 00 38 06 40 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 01 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 02 f7 e4 00 b3 17 6a 83 3f
00 00 00 00 e0 02 ff ff 47 21 00 00 02 04 05 a0
01 03 03 08 04 02 08 0a 00 41 d0 87 00 00 00 00
1d 10 3d 54 90 33 ec 3d 73 34 b6 4c 5e dd 03 9f
```

MAC:

```
90 33 ec 3d 73 34 b6 4c 5e dd 03 9f
```

6.1.2. Receive (server) SYN-ACK (covers options)

Receive_SYN_traffic_key:

```
e4 a3 7a da 2a 0a fc a8 71 14 34 91 3f e1 38 c7
71 eb cb 4a
```

IPv6/TCP:

```
6e 01 00 9e 00 38 06 40 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 02 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 01 00 b3 f7 e4 3f 51 99 4b
17 6a 83 40 e0 12 ff ff bf ec 00 00 02 04 05 a0
01 03 03 08 04 02 08 0a bd 33 12 9b 00 41 d0 87
1d 10 54 3d f1 cb a3 46 c3 52 61 63 f7 1f 1f 55
```

MAC:

```
f1 cb a3 46 c3 52 61 63 f7 1f 1f 55
```

6.1.3. Send (client) non-SYN (covers options)

Send_other_traffic_key:

```
1e d8 29 75 f4 ea 44 4c 61 58 0c 5b d9 0d bd 61
bb c9 1b 7e
```

IPv6/TCP:

```
6e 08 91 dc 00 73 06 40 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 01 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 02 f7 e4 00 b3 17 6a 83 40
3f 51 99 4c c0 18 01 00 32 9c 00 00 01 01 08 0a
00 41 d0 91 bd 33 12 9b 1d 10 3d 54 bf 08 05 fe
b4 ac 7b 16 3d 6f cd f2 ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff 00 43 01 04 fd e8 00 b4
01 01 01 79 26 02 06 01 04 00 01 00 01 02 02 80
00 02 02 02 00 02 02 42 00 02 06 41 04 00 00 fd
e8 02 08 40 06 00 64 00 01 01 00
```

MAC:

```
bf 08 05 fe b4 ac 7b 16 3d 6f cd f2
```

6.1.4. Receive (server) non-SYN (covers options)

Receive_other_traffic_key:

```
e4 a3 7a da 2a 0a fc a8 71 14 34 91 3f e1 38 c7
71 eb cb 4a
```

IPv6/TCP:

```
6e 01 00 9e 00 73 06 40 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 02 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 01 00 b3 f7 e4 3f 51 99 4c
17 6a 83 83 c0 18 01 00 ee 6e 00 00 01 01 08 0a
bd 33 12 a5 00 41 d0 91 1d 10 54 3d 6c 48 12 5c
11 33 5b ab 9a 07 a7 97 ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff 00 43 01 04 fd e8 00 b4
01 01 01 7a 26 02 06 01 04 00 01 00 01 02 02 80
00 02 02 02 00 02 02 42 00 02 06 41 04 00 00 fd
e8 02 08 40 06 00 64 00 01 01 00
```

MAC:

```
6c 48 12 5c 11 33 5b ab 9a 07 a7 97
```

6.2. SHA-1 MAC (omits TCP options)

6.2.1. Send (client) SYN (omits options)

Send_SYN_traffic_key:

```
31 a3 fa f6 9e ff ae 52 93 1b 7f 84 54 67 31 5c
27 0a 4e dc
```

IPv6/TCP:

```
6e 07 8f cd 00 38 06 40 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 01 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 02 c6 cd 00 b3 02 0c 1e 69
00 00 00 00 e0 02 ff ff a4 1a 00 00 02 04 05 a0
01 03 03 08 04 02 08 0a 00 9d b9 5b 00 00 00 00
1d 10 3d 54 88 56 98 b0 53 0e d4 d5 a1 5f 83 46
```

MAC:

```
88 56 98 b0 53 0e d4 d5 a1 5f 83 46
```

6.2.2. Receive (server) SYN-ACK (omits options)

Receive_SYN_traffic_key:

```
40 51 08 94 7f 99 65 75 e7 bd bc 26 d4 02 16 a2
c7 fa 91 bd
```

IPv6/TCP:

```
6e 0a 7e 1f 00 38 06 40 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 02 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 01 00 b3 c6 cd eb a3 73 4d
02 0c 1e 6a e0 12 ff ff 77 4d 00 00 02 04 05 a0
01 03 03 08 04 02 08 0a 5e c9 9b 70 00 9d b9 5b
1d 10 54 3d 3c 54 6b ad 97 43 f1 2d f8 b8 01 0d
```

MAC:

```
3c 54 6b ad 97 43 f1 2d f8 b8 01 0d
```

6.2.3. Send (client) non-SYN (omits options)

Send_other_traffic_key:

```
b3 4e ed 6a 93 96 a6 69 f1 c4 f4 f5 76 18 f3 65
6f 52 c7 ab
```

IPv6/TCP:

```
6e 07 8f cd 00 73 06 40 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 01 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 02 c6 cd 00 b3 02 0c 1e 6a
eb a3 73 4e c0 18 01 00 83 e6 00 00 01 01 08 0a
00 9d b9 65 5e c9 9b 70 1d 10 3d 54 48 bd 09 3b
19 24 e0 01 19 2f 5b f0 ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff 00 43 01 04 fd e8 00 b4
01 01 01 79 26 02 06 01 04 00 01 00 01 02 02 80
00 02 02 02 00 02 02 42 00 02 06 41 04 00 00 fd
e8 02 08 40 06 00 64 00 01 01 00
```

MAC:

```
48 bd 09 3b 19 24 e0 01 19 2f 5b f0
```

6.2.4. Receive (server) non-SYN (omits options)

Receive_other_traffic_key:

```
40 51 08 94 7f 99 65 75 e7 bd bc 26 d4 02 16 a2
c7 fa 91 bd
```

IPv6/TCP:

```
6e 0a 7e 1f 00 73 06 40 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 02 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 01 00 b3 c6 cd eb a3 73 4e
02 0c 1e ad c0 18 01 00 71 6a 00 00 01 01 08 0a
5e c9 9b 7a 00 9d b9 65 1d 10 54 3d 55 9a 81 94
45 b4 fd e9 8d 9e 13 17 ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff 00 43 01 04 fd e8 00 b4
01 01 01 7a 26 02 06 01 04 00 01 00 01 02 02 80
00 02 02 02 00 02 02 42 00 02 06 41 04 00 00 fd
e8 02 08 40 06 00 64 00 01 01 00
```

MAC:

```
55 9a 81 94 45 b4 fd e9 8d 9e 13 17
```

7. IPv6 AES-128 Output Test Vectors

AES-128 is computed as required by TCP-AO [RFC5926].

7.1. AES MAC (default - covers TCP options)

7.1.1. Send (client) SYN (covers options)

Send_SYN_traffic_key:

```
fa 5a 21 08 88 2d 39 d0 c7 19 29 17 5a b1 b7 b8
```

IP/TCP:

```
6e 04 a7 06 00 38 06 40 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 01 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 02 f8 5a 00 b3 19 3c cc ec
00 00 00 00 e0 02 ff ff de 5d 00 00 02 04 05 a0
01 03 03 08 04 02 08 0a 13 e4 ab 99 00 00 00 00
1d 10 3d 54 59 b5 88 10 74 81 ac 6d c3 92 70 40
```

MAC:

```
59 b5 88 10 74 81 ac 6d c3 92 70 40
```

7.1.2. Receive (server) SYN-ACK (covers options)

Receive_SYN_traffic_key:

```
cf 1b 1e 22 5e 06 a6 36 16 76 4a 06 7b 46 f4 b1
```

IPv6/TCP:

```
6e 06 15 20 00 38 06 40 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 02 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 01 00 b3 f8 5a a6 74 4e cb
19 3c cc ed e0 12 ff ff ea bb 00 00 02 04 05 a0
01 03 03 08 04 02 08 0a 71 da ab c8 13 e4 ab 99
1d 10 54 3d dc 28 43 a8 4e 78 a6 bc fd c5 ed 80
```

MAC:

```
dc 28 43 a8 4e 78 a6 bc fd c5 ed 80
```

7.1.3. Send (client) non-SYN (covers options)

Send_other_traffic_key:

```
61 74 c3 55 7a be d2 75 74 db a3 71 85 f0 03 00
```

IPv6/TCP:

```
6e 04 a7 06 00 73 06 40 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 01 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 02 f8 5a 00 b3 19 3c cc ed
a6 74 4e cc c0 18 01 00 32 80 00 00 01 01 08 0a
13 e4 ab a3 71 da ab c8 1d 10 3d 54 7b 6a 45 5c
0d 4f 5f 01 83 5b aa b3 ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff 00 43 01 04 fd e8 00 b4
01 01 01 79 26 02 06 01 04 00 01 00 01 02 02 80
00 02 02 02 00 02 02 42 00 02 06 41 04 00 00 fd
e8 02 08 40 06 00 64 00 01 01 00
```

MAC:

```
7b 6a 45 5c 0d 4f 5f 01 83 5b aa b3
```

7.1.4. Receive (server) non-SYN (covers options)

Receive_other_traffic_key:

```
cf 1b 1e 22 5e 06 a6 36 16 76 4a 06 7b 46 f4 b1
```

IPv6/TCP:

```
6e 06 15 20 00 73 06 40 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 02 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 01 00 b3 f8 5a a6 74 4e cc
19 3c cd 30 c0 18 01 00 52 f4 00 00 01 01 08 0a
71 da ab d3 13 e4 ab a3 1d 10 54 3d c1 06 9b 7d
fd 3d 69 3a 6d f3 f2 89 ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff 00 43 01 04 fd e8 00 b4
01 01 01 7a 26 02 06 01 04 00 01 00 01 02 02 80
00 02 02 02 00 02 02 42 00 02 06 41 04 00 00 fd
e8 02 08 40 06 00 64 00 01 01 00
```

MAC:

```
c1 06 9b 7d fd 3d 69 3a 6d f3 f2 89
```

7.2. AES MAC (omits TCP options)

7.2.1. Send (client) SYN (omits options)

Send_SYN_traffic_key:

```
a9 4f 51 12 63 e4 09 3d 35 dd 81 8c 13 bb bf 53
```

IPv6/TCP:

```
6e 09 3d 76 00 38 06 40 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 01 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 02 f2 88 00 b3 b0 1d a7 4a
00 00 00 00 e0 02 ff ff 75 ff 00 00 02 04 05 a0
01 03 03 08 04 02 08 0a 14 27 5b 3b 00 00 00 00
1d 10 3d 54 3d 45 b4 34 2d e8 bb 15 30 84 78 98
```

MAC:

```
3d 45 b4 34 2d e8 bb 15 30 84 78 98
```

7.2.2. Receive (server) SYN-ACK (omits options)

Receive_SYN_traffic_key:

```
92 de a5 bb c7 8b 1d 9f 5b 29 52 e9 cd 30 64 2a
```

IPv6/TCP:

```
6e 0c 60 0a 00 38 06 40 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 02 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 01 00 b3 f2 88 a6 24 61 45
b0 1d a7 4b e0 12 ff ff a7 0c 00 00 02 04 05 a0
01 03 03 08 04 02 08 0a 17 82 24 5b 14 27 5b 3b
1d 10 54 3d 1d 01 f6 c8 7c 6f 93 ac ff a9 d4 b5
```

MAC:

```
1d 01 f6 c8 7c 6f 93 ac ff a9 d4 b5
```

7.2.3. Send (client) non-SYN (omits options)

Send_other_traffic_key:

```
4f b2 08 6e 40 2c 67 90 79 ed 65 d4 bf 97 69 3d
```

IPv6/TCP:

```
6e 09 3d 76 00 73 06 40 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 01 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 02 f2 88 00 b3 b0 1d a7 4b
a6 24 61 46 c0 18 01 00 c3 6d 00 00 01 01 08 0a
14 27 5b 4f 17 82 24 5b 1d 10 3d 54 29 0c f4 14
cc b4 7a 33 32 76 e7 f8 ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff 00 43 01 04 fd e8 00 b4
01 01 01 79 26 02 06 01 04 00 01 00 01 02 02 80
00 02 02 02 00 02 02 42 00 02 06 41 04 00 00 fd
e8 02 08 40 06 00 64 00 01 01 00
```

MAC:

```
29 0c f4 14 cc b4 7a 33 32 76 e7 f8
```

7.2.4. Receive (server) non-SYN (omits options)

Receive_other_traffic_key:

```
92 de a5 bb c7 8b 1d 9f 5b 29 52 e9 cd 30 64 2a
```

IPv6/TCP:

```
6e 0c 60 0a 00 73 06 40 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 02 fd 00 00 00 00 00 00 00
00 00 00 00 00 00 00 01 00 b3 f2 88 a6 24 61 46
b0 1d a7 8e c0 18 01 00 34 51 00 00 01 01 08 0a
17 82 24 65 14 27 5b 4f 1d 10 54 3d 99 51 5f fc
d5 40 34 99 f6 19 fd 1b ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff 00 43 01 04 fd e8 00 b4
01 01 01 7a 26 02 06 01 04 00 01 00 01 02 02 80
00 02 02 02 00 02 02 42 00 02 06 41 04 00 00 fd
e8 02 08 40 06 00 64 00 01 01 00
```

MAC:

```
99 51 5f fc d5 40 34 99 f6 19 fd 1b
```

8. Observed Implementation Errors

The following is a partial list of implementation errors that this set of test vectors is intended to validate.

8.1. Algorithm issues

- Underlying implementation of HMAC SHA1 or AES128 CMAC does not pass their corresponding test vectors [RFC2202] [RFC4493]
- SNE algorithm does not consider corner cases (the pseudocode in [RFC5925] was not intended as complete, as discussed in [To20])

8.2. Algorithm parameters

- KDF context length is incorrect, e.g. it does not include TCP header length + payload length (it should, per 5.2 of TCP-AO [RFC5925])
- KDF calculation does not start from counter i = 1 (it should, per Sec. 3.1.1 of the TCP-AO crypto algorithms [RFC5926])
- KDF calculation does not include output length in bits, contained in two bytes in network byte order (it should, per Sec. 3.1.1 of the TCP-AO crypto algorithms [RFC5926])
- KDF uses keys generated from current TCP segment sequence numbers (KDF should use only local and remote ISNs or zero, as indicated in Sec. 5.2 of TCP-AO [RFC5925])

8.3. String handling issues

The strings indicated in TCP-AO and its algorithms are indicated as a sequence of bytes of known length. In some implementations, string lengths are indicated by a terminal value (e.g., zero in C). This terminal value is not included as part of the string for calculations.

- Password includes the last zero-byte (it should not)
- Label "TCP-AO" includes the last zero byte (it should not)

8.4. Header coverage issues

- TCP checksum and/or MAC is not zeroed properly before calculation (both should be)
- TCP header is not included to the MAC calculation (it should be)
- TCP options are not included to the MAC calculation by default (there is a separate parameter in the master key tuple to ignore options; this document provides test vectors for both options-included and options-excluded cases)

9. Security Considerations

This document is intended to assist in the validation of implementations of TCP-AO, to further enable its more widespread use as a security mechanism to authenticate not only TCP payload contents but the TCP headers and protocol.

The master_key of "testvector" used here for test vector generation SHOULD NOT be used operationally.

10. IANA Considerations

This document contains no IANA issues. This section should be removed upon publication as an RFC.

11. References

11.1. Normative References

[RFC791] Postel, J., "Internet Protocol," RFC 791, Sept. 1981.

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

[RFC2018] Mathis, M., J. Mahdavi, S. Floyd, A. Romanow, "TCP Selective Acknowledgment Options," RFC 2018, Oct. 1996.

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

[RFC5925] Touch, J., A. Mankin, R. Bonica, "The TCP Authentication Option," RFC 5925, June 2010.

[RFC5926] Lebovitz, G., and E. Rescorla, "Cryptographic Algorithms for the TCP Authentication Option (TCP-AO)," RFC 5925, June 2010.

[RFC6978] Touch, J., "A TCP Authentication Option Extension for NAT Traversal," RFC 6978, July 2013.

[RFC7323] Borman, D., B. Braden, V. Jacobson, R. Scheffenegger, Ed., "TCP Extensions for High Performance," RFC 7323, Sept. 2014.

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

[RFC8200] Deering, S., R. Hinden, "Internet Protocol Version 6 (IPv6) Specification," RFC 8200, Jul. 2017.

11.2. Informative References

[RFC2202] Cheng, P., and R. Glenn, "Test Cases for HMAC-MD5 and HMAC-SHA-1," RFC 2202, Sept. 1997.

[RFC4493] Song, JH, R. Poovendran, J. Lee, T. Iwata, "The AES-CMAC Algorithm," RFC 4493, June 2006.

[To20] Touch, J., "Sequence Number Extension for Windowed Protocols," draft-tsvwg-touch-sne, Jun. 2020.

12. Acknowledgments

(TBD)

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

Juhamatti Kuusisaari
Infinera Corporation
Sinimaentie 6c
FI-02630 Espoo, Finland
Email: jkuusisaari@infinera.com