Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 9, 2017

M. Bagnulo
UC3M
K. De Schepper
Nokia Bell Labs
G. Judd
Morgan Stanley
July 8, 2016

Recommendations for increasing TCP performance in low RTT networks. draft-bagnulo-tcpm-tcp-low-rtt-00

Abstract

This documents compiles a set of issues that negatively affect TCP performance in low RTT networks as well as the recommendations to overcome them.

Status of This Memo

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

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

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

This Internet-Draft will expire on January 9, 2017.

Copyright Notice

Table of Contents

1. Introduction

Over the last few years there has been significant operational experience about running TCP in networks with low RTTs. By networks with low RTTs we mean networks with RTTs between a few microsecs and a few hundreds of microsecs. These networks are typically found in datacenters and in addition to a low RTT they usually exhibit a high bandwidth (tens of Gbps). there are a number of reports and papers that show that TCP performance in such environment can be poor and that TCP needs to be tuned and even updated to provide good performance. The goal of this memo is to summarize the set of changes needed to TCP to perform well in these environments.

There are transport protocols, notably DCTCP [I-D.ietf-tcpm-dctcp] that have been specifically designed to perform well in data center environments where low RTT is the norm. However, due to several reasons, many datacenters also need to need to use TCP for their communications (see section 7.1 of [judd-nsdi] for the motivation for using TCP in a production datacenter). This is the reason why the recommendations about how to update TCP to run in these environments are relevant. Some of the recommendations contained in this note may also apply to protocols such as DCTCP, but the main goal of this note is TCP.

We next describe different issues that have been identified and the changes that would be required in the TCP specifications and/or the TCP implementations to address them.

2. Minimum Retransmission timer

Current TCP specification recommend that the minimum retransmission timer (RTOmin) should be at least 1 second. According to [incast], current implementations, RTOmin is set between 200 ms and 400 ms. In a network with RTT in the order of microseconds, this imposes large periods of inactivity when a packet is lost and its loss is detected via the retransmission timeout. This also aggravates the so called TCP incast problem. This issues has been reported in several papers, including [incast-wren], [judd-nsdi], and [incast]. One proposed mitigation to this problem that results in better performance is to reduce RTOmin.

From a specification perspective, RFC 6298 [RFC6298] states that:

> (2.4) Whenever RTO is computed, if it is less than 1 second, then the RTO SHOULD be rounded up to 1 second.

[incast] suggests that using RTOmin equal to 200 microsecs provides significant performance improvement in terms of goodput and that even no RTOmin results in even better performance.

Using a lower RTOmin while it goes against the recommendation included in RFC6298, it is supported as the specification as the RTOmin of 1 ms is not mandatory, just a recommendation. However, it would beneficial to update RFC6298 in this aspect and to provide a recommendation (maybe in the form of BCP) that for low RTT networks, a smaller RTOmin should be used.

This has an implication on the clock granularity when calculating RTO. RFC6298 doe not impose any requirement on the granularity of the clock used to measure the RTT used for the RTO calculation. It does state that finer clock granularities (below 100 ms) perform better. In order to achieve RTOmin of 200 micro secs or less, the granularity must be finer than the the RTOmin allowed. According to [incast] and [judd-nsdi] current linux systems can achieve a RTOmin of 4 ms due to the coarse granularity. so, providing a recommendation in terms of the granularity may also be useful.

3. Delay for Delayed ACKs

[judd-nsdi] reports that the default value for the delay for delayed ACKs ranges between tens and hundreds of ms. For low RTTs, a lower value of delay achieves a higher performance (see [judd-nsdi]) and hence a value of 1 ms or lower should be recommended for low RTT networks.

From a specification perspective, current specifications do not require a minimum waiting time for generating the delayed ACKs. They do impose a maximum waiting time. In particular, RFC 1122 [RFC1122] states that:

> A TCP SHOULD implement a delayed ACK, but an ACK should not be excessively delayed; in particular, the delay MUST be less than 0.5 seconds, and in a stream of full-sized segments there SHOULD be an ACK for at least every second segment.

Also, RFC5681 [RFC5681] states that:

> The delayed ACK algorithm specified in [RFC1122] SHOULD be used by a TCP receiver. When using delayed ACKs, a TCP receiver MUST NOT excessively delay acknowledgments. Specifically, an ACK SHOULD be generated for at least every second full-sized segment, and MUST be generated within 500 ms of the arrival of the first unacknowledged packet.

So, from a specification perspective, current RFCs do not need to be updated, but it may be useful to provide a recommendation in the form of BCP that for low RTT environments, the delay used for delayed ACKs should be tuned accordingly.

4. Minimum Congestion window

Current specifications require that the minimum congestion window is 2MSS. As pointed out in [TCP-sub-mss-w] and [judd-nsdi], in the case of small RTTs, this may result in a considerably large rate, below which TCP becomes unresponsive to congestion. In particular, with a SMSS of 1500 B and a RTT of 50 micro secs, this results in a rate of 240Mbps.

In terms of specifications, according to RFC5681, the CWND in Fast Retransmit and Fast Recovery is calculated as:

> 2. When the third duplicate ACK is received, a TCP MUST set ssthresh to no more than the value given in equation (4).
>
> 6. When the next ACK arrives that acknowledges previously unacknowledged data, a TCP MUST set cwnd to ssthresh (the value set in step 2). This is termed "deflating" the window.
>
> ssthresh = max (FlightSize / 2, 2*SMSS) (4)

In order to address this issue, it is necessary to modify TCP behaviour to function with CWND smaller than 2 MSS. This would require a update to RFC 5681. Several possibilities have been

proposed to accommodate this need. [TCP-sub-mss-w] and [TCP-nice] propose possible solutions.

5. Other issues

[judd-nsdi] identifies that in networks where the propagation delay and the transmission delay are very small, the queuing delay affects the RTT severely resulting in significant changes in the RTT. This has a negative effect in the calculation of the receiver buffer when using autotunning, since the buffer is calculated using the RTT estimation. The result is that it is frequent in these scenarios that the TCP connection is limited by the receiver buffer/RCVWND.

As far i can tell, there is no RFC that defines how to calculate the receive buffer, so no change in any spec would be required to address this, but maybe it is worthwhile to define a mechanism for autotunning for small RTTs and/or to do some recommendation in this regard.

6. Concluding remarks

This document compiles a number of issues that have been previously identified as harming TCP performance in low RTT networks. Some of the issues require updates in the current specifications and probably most of the issues may deserve some form of recommendation in the form of a BCP for using TCP in low RTT networks. It may make sense to work on the changes in the specification and the definition of new specifications (in particular for the case of lower than 1 MSS CWND) and then evolve this document to become the BCP for low RTT environments.

7. Security considerations

TBD, not sure if there is any.

8. IANA Considerations

There are no IANA considerations in this memo.

9. Acknowledgments

TBD

10. Informative References

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

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

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

[I-D.ietf-tcpm-dctcp] Bensley, S., Eggert, L., Thaler, D., Balasubramanian, P., and G. Judd, "Datacenter TCP (DCTCP): TCP Congestion Control for Datacenters", draft-ietf-tcpm-dctcp-01 (work in progress), November 2015.

[judd-nsdi] Judd, G., "Attaining the promise and avoiding the pitfalls of TCP in the Datacenter", NSDI 2015, 2015.

[incast] V., V., A., A., H., H., E., E., D., D., G., G., G., G., and B. B., "Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication", ACM SIGCOMM 2009, 2009.

[incast-wren] Y., Y., R., R., J., J., R., R., and A. A., "Understanding TCP Incast Throughput Collapse in Datacenter Networks", WREN 2009, 2009.

[TCP-nice] A., A., R., R., M., M., R., R., and A. A., "TCPNICE: A mechanism for background transfers", SIGOPS Oper. Syst. Review 2002, 2002.

[TCP-sub-mss-w] Briscoe, B. and K. De Schepper, "Scaling TCP's Congestion Window for Small Round Trip Times", BT Technical Report TR-TUB8-2015-002, May 2015, <http://www.bobbriscoe.net/projects/latency/sub-mss-w.pdf>.

Authors' Addresses

Marcelo Bagnulo
Universidad Carlos III de Madrid
Av. Universidad 30
Leganes, Madrid 28911
SPAIN

Phone: 34 91 6249500
Email: marcelo@it.uc3m.es
URI: http://www.it.uc3m.es

Koen De Schepper
Nokia Bell Labs
Antwerp
Belgium

Email: koen.de_schepper@nokia.com
URI: https://www.bell-labs.com/usr/koen.de_schepper

Glenn Judd
Morgan Stanley

Email: Glenn.Judd@MorganStanley.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 9, 2017

M. Bagnulo
UC3M
B. Briscoe
Simula Research Lab
July 8, 2016

Adding Explicit Congestion Notification (ECN) to TCP control packets
draft-bagnulo-tsvwg-generalized-ecn-01

Abstract

This documents explores the possibility of adding ECN support to TCP control packets.

Status of This Memo

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

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

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

This Internet-Draft will expire on January 9, 2017.

Copyright Notice

Table of Contents

1. Introduction

RFC3168 [RFC3168] specifies the support of Explicit Congestion Notification (ECN) to IP. By using the ECN capability, switches performing Active Queue Management (AQM) can use ECN marks instead of packets drops to signal congestion to the endpoints of a communication. This results in lower packet loss and increased performance. However, RFC3168 specifies the support of ECN in TCP data packets, but precludes the use of ECN in TCP control packets (TCP SYN, TCP SYN/ACK, pure ACKs, Window probes) and in retransmitted packets. RFC 5562 [RFC5562] is an experimental extension to ECN that enables the ECN support for TCP SYN/ACK packets.

The inability of using ECN in TCP control packets has a potential harmful effect, especially in environments where ECN support is pervasive. For example, [judd-nsdi] shows that in a data center environment where DCTCP is used (in conjunction with ECN), the the probability of being able to establish a new connection using a non-ECT-marked SYN packet drops to close to 0 when there are 16 ongoing TCP flows transmitting at full speed. In this particular context of a datacenter using DCTCP, the issue is that the proposed AQM aggressively marks packets to keep the buffer queues small and this implies that non-ECT-marked packets are in turn dropped aggressively as well, rendering nearly impossible to establish new connection when there is ongoing traffic.

These limitations are not limited to the data center environment. In any ECN deployment, non ECT marked packets suffer a penalty when they traverse a congested bottleneck. For instance, with a drop probability of 1%, 1% of connection attempts suffer a timeout before the SYN is retransmitted, which is very deterimental to the performance of short flows. Dropping TCP control traffic, such as TCP SYNs and pure ACKs have a negative effect on the overall performance of the communication, so it is beneficial to avoid it.

Finally, there are ongoing efforts to promote the adoption of DCTCP (and similar transports) over the Internet to achieve low latency for all communications [I-D.briscoe-tsvwg-aqm-tcpm-rmcat-l4s-problem]. In such approach, ECN capable packets are treated more favorably, as they are likely to experience less delay and lower packet drop probability. Preventing TCP control packets, which are critical for TCP performance, to obtain the benefits of ECN would result in degraded performance.

However, RFC3168 does not prevents from using ECN in TCP control packets lightly. It provides a number of specific reasons for each packet type. In this note, we revisit each of the arguments provided by RFC3168 and explore possibilities to enable the ECN capability in the different packet types. We do so in the context of a data center network and in the context of the public Internet.

2. The reliability argument

While for each type of packet RFC 3168 provides a set of specific arguments for preventing their marking, RFC3168 presents the reliable delivery of the congestion signal as an overarching argument that needs to be consider when trying to enable the ECT marking of TCP control packets. In particular, Section 5.2 of RFC3168 states:

> To ensure the reliable delivery of the congestion indication of the CE codepoint, an ECT codepoint MUST NOT be set in a packet unless the loss of that packet in the network would be detected by the end nodes and interpreted as an indication of congestion.

We believe this argument is overly conservative. The overall principle that should determine the level of reliability required for ECN capable packets should be the one of "do not harm". Reliable delivery of the CE codepoint is indeed paramount but the level of reliability required should be the one of the original congestion signal (i.e. the detection of the loss of the original packet). In other words, the situation without ECN is that when a packet is to be transmitted through a congested link, the packet may be dropped and that is the congestion signal sent to the endpoint. When ECN is introduced, the reliability of the delivery of the congestion signal should be no worse than without ECN. In particular, setting the CE codepoint in the very same packet seem to fulfill this criteria, since either the packet is delivered and the CE codepoint signal is delivered to the endpoint, or the packet is dropped, so the original congestion signal through the packet loss is delivered to the endpoint. Requiring more than this implies that the ECN congestion signal is delivered more reliably than the current situation, which is not a bad thing per se, but, as we describe in this memo, it

results in performance penalties that should be reconsidered in the view of current deployments.

In addition, the reliability of the delivery of the congestion signal is used an argument for not setting the ECT codepoint in TCP control packets, which effectively reduced the reliability of the transmission of these TCP control packets. There is the then a tradeoff between the reliability of the delivery of the congestion signal and the reliability of the delivery of TCP control packets. As currently specified, ECN adoption implies an increased reliability of the ECN congestion signal and a decrease in the reliability in the TCP control packets. We believe that it is possible and desirable to restore the tradeoff existent in non ECN capable networks in terms of reliability, where the congestion signal delivery is as reliable as in a non ECN capable network and so it is the delivery of TCP control packets.

3. TCP SYNs

We next describe he arguments exhibited by current specification for precluding the ECT marking of SYN packets.

In addition to the reliability argument above, RFC 5562 presents two arguments against ECT marking of SYN packets (cited verbatim):

> There are several reasons why an ECN-Capable codepoint must not be set in the IP header of the initiating TCP SYN packet. First, when the TCP SYN packet is sent, there are no guarantees that the other TCP endpoint (node B in Figure 2) is ECN-Capable, or that it would be able to understand and react if the ECN CE codepoint was set by a congested router.
>
> Second, the ECN-Capable codepoint in TCP SYN packets could be misused by malicious clients to "improve" the well-known TCP SYN attack. By setting an ECN-Capable codepoint in TCP SYN packets, a malicious host might be able to inject a large number of TCP SYN packets through a potentially congested ECN-enabled router, congesting it even further.

We next go through all the arguments stated above to enable ECT marking of SYN packets.

Argument 1: Unknown ECN capability capability at the responder. The initiator does not know whether the responder supports ECN and in particular, the initiator does not know if the responder supports ECT marked SYNs.

In the DC context, this argument does not hold (at least in single tenant DCs, possibly in multi-tenant DCs, if we assume that each tenant mostly communicates with its own VMs). The DC is a much more controlled environment than the public Internet, so the server's support of ECN can be guaranteed administratively i.e. the manager of the DC makes sure that the servers support ECN and in particular ECT marked SYN packets.

In the public Internet context, it cannot be assumed that all servers support ECN, and much less that they support ECT marked SYN packets. When sending an ECT marked SYN to a legacy responder (i.e. a responder that does not support ECT marked SYNs), different behaviours are possible.

The responder may drop the SYN (either silently or by sending a RST) or may reply with a non ECT marked SYN/ACK. If it is the latter, then this is a non-issue (the second issue presented next still applies though). If it is the former, then the initiator will have to retransmit the SYN (without the ECT mark). Depending how extended is this behaviour, this can reduce significantly the benefits of adding ECT capability to the SYN or even be detrimental for the performance. According to [ecn-pam], out of the top 1M Alexa web sites, 0,82% of IPv4 sites and 0,61% of IPv6 sites fail to establish a connection when they receive a TCP SYN with any ECN codepoint set.

If based on this data, we conclude that the fraction of fraction of servers that discard the ECT marked SYN is a non negligible, further options depend on whether they silently discard it or they send a RST back. If they send a RST back, the initiator can then send a non ECT marked SYN. In this case the penalty would be an extra RTT, which may or may not be acceptable, depending on the fraction of servers that behaves like this. If the server silently discard the ECT marked SYN, then the initiator needs to wait for the retransmission timer to expire and retransmit a non-ECT marked SYN. This is a high penalty. If this is the case, one option, would be to first send an ECT marked SYN and then a non-ECT marked SYN (possibly with a small delay between them) and establish the ECT capable connection if the former is replied. But it is questionable whether the level of failure of ECT on SYNs warrants this, particularly given failures could reduce if ECN on SYNs is standardized.

Argument 2: Loss of congestion notification in the SYN packet due to lack of support from the responder. If the ECT marked SYN packet is tagged as CE by a router along the path and the server does not support ECT marked SYN packets, even if the server replies with a SYN/ACK, the congestion information would be lost.

The accurate ECN (AccECN) proposal [I-D.ietf-tcpm-accurate-ecn] suggests a two-pringed solutions to this problem. First AccECN provides a way for the responder to feedback whether there was CE on the SYN, and second AccECN introduces a different combination of TCP header flags on the SYN/ACK so that the initiator knows whether or not the responder supports AccECN. Then if the responder does indicate that it supports AccECN the initiator can be sure that, if there is no CE feedback on the SYNACK, then there really was no CE on the SYN.

If the responder's SYN/ACK shows that it does not support AccECN, the initiator can take a conservative approach and assume the SYN was marked with CE and reduce its initial window. However, the initiator knows that congestion is not serious, because both the SYN and the SYN/ACK were delivered through the network. Therefore congestion is not serious enough for a router to have had to turn off ECN. Therefore, even a conservative initiator would not have to reduce its initial window as much as it would in response to a timeout following no response to its SYN.

Nonetheless, even a slight conservative reduction in initial window might be a significant penalty, especially in the early days of deployment, when little support for ECT SYN packets will be available. This could be mitigated by caching previous experience of which servers support AccECN.

Argument 3: DoS attacks. There are two possible DoS attacks involved in the text contained in RFC3168. On one hand, the mention about improving the well-known TCP SYN attack. The reference to the TCP SYN attack we interpret it as a reference to the TCP SYN flood attack (see https://en.wikipedia.org/wiki/SYN_flood). This attack is addressed to the responder endpoint of the connection. The argument is basically, because SYN can be used to launch attacks, their transmission should not be more reliable. While it is true that SYNs can be used to launch attacks, it is also true that SYNs are fundamental for legitimate communications, so the argument for increasing reliability of legitimate communications should take precedence. On the other hand in the RFC3168 refers about ECN capable SYN packets to congest further a bottleneck. It is not clear why a TCP SYN packet is worse than any other packet in this respect. In any case, section 7 of RFC3168 already provides the means to address this concern, as it reads:

> First, ECN-Capable routers will only mark packets (as opposed to dropping them) when the packet marking rate is reasonably low. During periods where the average queue size exceeds an upper threshold, and therefore the potential packet marking rate would

> be high, our recommendation is that routers drop packets rather then set the CE codepoint in packet headers.
>
> Safe deployment of ECN requires that network devices drop excessive traffic, even when marked as originating from an ECN-capable transport. This is a necessary safety precaution because:..

Alternative behaviour. If we were to allow setting the ECT codepoint in the SYN packets, we need to define how it would behave.

One challenge is to support legacy ECN responders that do not support ECT marked SYNs but do support ECN.

One possible behaviour could be something along these lines. The SYN packet will carry the ECT(1) bit set as well as the ECE and CWR bits set. This is needed to support legacy ECN responders that would ignore the ECT bit, but properly process the ECN support negotiation using the ECE and CWR flags. Routers can then set the CE bit in the SYN.

If the responder receives a SYN with ECT(1), ECE and CWR bits set, it replies with a SYN/ACK that includes ECT(1) bit set. Because the ECT(1) bit is set, (and the CWR bit is not set) the initiator can realize that the responder supports ECN and also ECT marked SYNs.

If the responder receives a SYN with ECT(1), ECE, CWR and CE bits set, it replies with a SYN/ACK that includes the ECT(1) and the ECE bits set. Because the ECT(1) bit is set (and the CWR bit is not set), the initiator can realize that the ECE bit means that the CE bit was set in the SYN and then can react accordingly. The reaction to the ECE bit is then to halve the initial CWND for the connection.

4. Pure ACKs.

RFC3168 exposes the following arguments for not allowing the ECT marking of pure ACKs. In section 5.2 it reads:

> To ensure the reliable delivery of the congestion indication of the CE codepoint, an ECT codepoint MUST NOT be set in a packet unless the loss of that packet in the network would be detected by the end nodes and interpreted as an indication of congestion.
>
> Transport protocols such as TCP do not necessarily detect all packet drops, such as the drop of a "pure" ACK packet; for example, TCP does not reduce the arrival rate of subsequent ACK packets in response to an earlier dropped ACK packet. Any proposal for extending ECN- Capability to such packets would have

> to address issues such as the case of an ACK packet that was marked with the CE codepoint but was later dropped in the network. We believe that this aspect is still the subject of research, so this document specifies that at this time, "pure" ACK packets MUST NOT indicate ECN-Capability.

Later on, in section 6.1.4 it reads:

> For the current generation of TCP congestion control algorithms, pure acknowledgement packets (e.g., packets that do not contain any accompanying data) MUST be sent with the not-ECT codepoint. Current TCP receivers have no mechanisms for reducing traffic on the ACK-path in response to congestion notification. Mechanisms for responding to congestion on the ACK-path are areas for current and future research. (One simple possibility would be for the sender to reduce its congestion window when it receives a pure ACK packet with the CE codepoint set). For current TCP implementations, a single dropped ACK generally has only a very small effect on the TCP's sending rate.

We next address each of the arguments presented above.

The first argument is about lack of reliability while conveying congestion notification information when carried in pure ACKs. This is the specific instance for the pure ACK messages of the reliability argument discussed in Section 2. In some cases, the loss of pure ACKs is not detected by the endpoints, loosing the congestion notification information indadvertedly if it was to be carried in those packets. As we argued before, the bar for deciding if a packet can be marked with the ECT codepoint i.e. if it is suitable for carrying congestion notification information is that the congestion signal communication should be as reliable as dropping the packet. After all, the alternative of setting the CE bit in the packet is dropping the packet. So, the question is whether carrying congestion information in a pure ACK conveys the congestion information as reliably as when the pure ACK is dropped and it is obvious that the answer to that question is clearly yes. If the pure ACK carrying the ECT and the CE bits set is later dropped by the network, it will be essentially falling back to the use of drop as congestion signal.

The second argument exhibited in RFC3168 is the lack of means in the sender of the pure ACKs to reduce the load that is creating the congestion. Again, marking the pure ACKs with the ECT codepoint and allowing them to carry congestion notification information would be no worse than not doing so from this perspective (and it would be much more detrimental form the overall performance perspective). The sender of the pure ACKs will receive the echo of the congestion notification and it may be able to reduce the CWND of the connection.

If it happens to be only sending pure ACKs and no data and it can react reducing the rate at which data is being sent, it would not be worse in terms of congestion than in the case that the pure ACK is dropped.

So, overall, we believe that in terms of conveying and reacting to congestion, allowing to set the ECT (and the CE) flags in the pure ACKs is not worse than not doing so (and dropping the pure ACK), but in terms of performance, not ECT marking the pure ACKs is certainly detrimental.

5. Retransmitted packets.

RFC3168 does not allow setting the ECT codepoint in retransmitted packets. The arguments presented in the specification for supporting this design choice are the following ones (the text is quite long, not sure if we should keep it all):

> This document specifies ECN-capable TCP implementations MUST NOT set either ECT codepoint (ECT(0) or ECT(1)) in the IP header for retransmitted data packets, and that the TCP data receiver SHOULD ignore the ECN field on arriving data packets that are outside of the receiver's current window. This is for greater security against denial-of-service attacks, as well as for robustness of the ECN congestion indication with packets that are dropped later in the network.
>
> First, we note that if the TCP sender were to set an ECT codepoint on a retransmitted packet, then if an unnecessarily-retransmitted packet was later dropped in the network, the end nodes would never receive the indication of congestion from the router setting the CE codepoint. Thus, setting an ECT codepoint on retransmitted data packets is not consistent with the robust delivery of the congestion indication even for packets that are later dropped in the network.
>
> In addition, an attacker capable of spoofing the IP source address of the TCP sender could send data packets with arbitrary sequence numbers, with the CE codepoint set in the IP header. On receiving this spoofed data packet, the TCP data receiver would determine that the data does not lie in the current receive window, and return a duplicate acknowledgement. We define an out-of-window packet at the TCP data receiver as a data packet that lies outside the receiver's current window. On receiving an out-of-window packet, the TCP data receiver has to decide whether or not to treat the CE codepoint in the packet header as a valid indication of congestion, and therefore whether to return ECN-Echo indications to the TCP data sender. If the TCP data receiver

ignored the CE codepoint in an out-of-window packet, then the TCP data sender would not receive this possibly- legitimate indication of congestion from the network, resulting in a violation of end-to-end congestion control. On the other hand, if the TCP data receiver honors the CE indication in the out-of-window packet, and reports the indication of congestion to the TCP data sender, then the malicious node that created the spoofed, out-of- window packet has successfully "attacked" the TCP connection by forcing the data sender to unnecessarily reduce (halve) its congestion window. To prevent such a denial-of-service attack, we specify that a legitimate TCP data sender MUST NOT set an ECT codepoint on retransmitted data packets, and that the TCP data receiver SHOULD ignore the CE codepoint on out-of-window packets.

One drawback of not setting ECT(0) or ECT(1) on retransmitted packets is that it denies ECN protection for retransmitted packets. However, for an ECN-capable TCP connection in a fully-ECN-capable environment with mild congestion, packets should rarely be dropped due to congestion in the first place, and so instances of retransmitted packets should rarely arise. If packets are being retransmitted, then there are already packet losses (from corruption or from congestion) that ECN has been unable to prevent.

We note that if the router sets the CE codepoint for an ECN-capable data packet within a TCP connection, then the TCP connection is guaranteed to receive that indication of congestion, or to receive some other indication of congestion within the same window of data, even if this packet is dropped or reordered in the network. We consider two cases, when the packet is later retransmitted, and when the packet is not later retransmitted.

In the first case, if the packet is either dropped or delayed, and at some point retransmitted by the data sender, then the retransmission is a result of a Fast Retransmit or a Retransmit Timeout for either that packet or for some prior packet in the same window of data. In this case, because the data sender already has retransmitted this packet, we know that the data sender has already responded to an indication of congestion for some packet within the same window of data as the original packet. Thus, even if the first transmission of the packet is dropped in the network, or is delayed, if it had the CE codepoint set, and is later ignored by the data receiver as an out- of-window packet, this is not a problem, because the sender has already responded to an indication of congestion for that window of data.

In the second case, if the packet is never retransmitted by the data sender, then this data packet is the only copy of this data

> received by the data receiver, and therefore arrives at the data receiver as an in-window packet, regardless of how much the packet might be delayed or reordered. In this case, if the CE codepoint is set on the packet within the network, this will be treated by the data receiver as a valid indication of congestion.

There are essentially three arguments for not ECT marking retransmitted packets, namely, reliability, DoS attacks and over-reaction to congestion. We address all of them next in order.

About reliability, as described in Section 2, we believe that the bar should be that the congestion signal should be delivered as reliably as if it was a packet drop. So, if a retransmitted packet is dropped and this goes by unnoticed by the receiver, then the congestion signal expressed as a drop would be lost. The same applies to the congestion signal resulting from marking with ECT and CE the very same retransmitted packet which later is dropped.

About the possibility of DoS attacks, the protection against the DoS attack does not result from not allowing retransmitted packets to be ECT marked. If an attacker decided to launch such an attack, it would craft the packet with the ECT codepoint set. Effectively, the protection against the described DoS attack comes from the requirement that the receiver should not ignore the CE codepoint in out-of-window packets. We proposed to allow ECT marking of retransmitted packets, in order reduces the chances of it being dropped, but keep the requirement to ignore the CE codepoint in out-of-window packets.

Finally, the third argument is about over-reacting to congestion. Basically, if the retransmitted packet is dropped, the sender will not react again to congestion (it has reacted already when it generated the retransmitted packet). If the retransmitted packet is CE tagged instead of dropped, then the congestion signal will arrive again to the sender who could potentially react again to congestion. However, this should not happen as RFC3168 imposes the condition that a sender must only react once per window to the congestion signal and this should not be an exception to this rule.

6. Window probe packets

RFC3168 presents only the reliability argument for preventing setting the ECT codepoint in Window Probe packets. Specifically, it states:

> When the TCP data receiver advertises a zero window, the TCP data sender sends window probes to determine if the receiver’s window has increased. Window probe packets do not contain any user data except for the sequence number, which is a byte. If a window

probe packet is dropped in the network, this loss is not detected by the receiver. Therefore, the TCP data sender MUST NOT set either an ECT codepoint or the CWR bit on window probe packets.

However, because window probes use exact sequence numbers, they cannot be easily spoofed in denial-of-service attacks. Therefore, if a window probe arrives with the CE codepoint set, then the receiver SHOULD respond to the ECN indications.

The reliability argument has been addressed in Section 2. dropping the window probe message in the case the conditions for the Silly Window Syndrome are on, basically implies that the sender will be stalled until the new Window Probe message reaches the receiver, which agains results in a performance penalty.

On the bright side, receivers should respond to ECN messages in these packets, so changing the behaviour should be less painful than for other packet types.

7. Security considerations

TBD, not sure if there is any.

8. IANA Considerations

There are no IANA considerations in this memo.

9. Acknowledgments

TBD

10. Informative References

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

[RFC5562] Kuzmanovic, A., Mondal, A., Floyd, S., and K. Ramakrishnan, "Adding Explicit Congestion Notification (ECN) Capability to TCP's SYN/ACK Packets", RFC 5562, DOI 10.17487/RFC5562, June 2009, <http://www.rfc-editor.org/info/rfc5562>.

[I-D.briscoe-tsvwg-aqm-tcpm-rmcat-l4s-problem]
Briscoe, B., Schepper, K., and M. Bagnulo, "Low Latency, Low Loss, Scalable Throughput (L4S) Internet Service: Problem Statement", draft-briscoe-tsvwg-aqm-tcpm-rmcat-l4s-problem-02 (work in progress), July 2016.

[I-D.ietf-tcpm-accurate-ecn]
Briscoe, B., Kuehlewind, M., and R. Scheffenegger, "More Accurate ECN Feedback in TCP", draft-ietf-tcpm-accurate-ecn-01 (work in progress), June 2016.

[judd-nsdi]
Judd, G., "Attaining the promise and avoiding the pitfalls of TCP in the Datacenter", NSDI 2015, 2015.

[ecn-pam] Brian, B., Mirja, M., Damiano, D., Iain, I., Gorry, G., and R. Richard, "Enabling Internet-Wide Deployment of Explicit Congestion Notification", PAM 2015, 2015.

Authors' Addresses

Marcelo Bagnulo
Universidad Carlos III de Madrid
Av. Universidad 30
Leganes, Madrid 28911
SPAIN

Phone: 34 91 6249500
Email: marcelo@it.uc3m.es
URI: http://www.it.uc3m.es

Bob Briscoe
Simula Research Lab

Email: ietf@bobbriscoe.net
URI: http://bobbriscoe.net/

TCP Maintenance Working Group
Internet-Draft
Intended status: Experimental
Expires: January 7, 2017

Y. Cheng
N. Cardwell
Google, Inc

July 6, 2016

RACK: a time-based fast loss detection algorithm for TCP
draft-cheng-tcpm-rack-01

Abstract

This document presents a new TCP loss detection algorithm called RACK ("Recent ACKnowledgment"). RACK uses the notion of time, instead of packet or sequence counts, to detect losses, for modern TCP implementations that can support per-packet timestamps and the selective acknowledgment (SACK) option. It is intended to replace the conventional DUPACK threshold approach and its variants, as well as other nonstandard approaches.

Status of This Memo

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

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

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

This Internet-Draft will expire on January 7, 2017.

Copyright Notice

1. Introduction

This document presents a new loss detection algorithm called RACK ("Recent ACKnowledgment"). RACK uses the notion of time instead of the conventional packet or sequence counting approaches for detecting losses. RACK deems a packet lost if some packet sent sufficiently later has been delivered. It does this by recording packet transmission times and inferring losses using cumulative acknowledgments or selective acknowledgment (SACK) TCP options.

In the last couple of years we have been observing several increasingly common loss and reordering patterns in the Internet:

1. Lost retransmissions. Traffic policers [POLICER16] and burst losses often cause retransmissions to be lost again, severely increasing TCP latency.

2. Tail drops. Structured request-response traffic turns more losses into tail drops. In such cases, TCP is application-limited, so it cannot send new data to probe losses and has to rely on retransmission timeouts (RTOs).

3. Reordering. Link layer protocols (e.g., 802.11 block ACK) or routers' internal load-balancing can deliver TCP packets out of order. The degree of such reordering is usually within the order of the path round trip time.

Despite TCP stacks (e.g. Linux) that implement many of the standard and proposed loss detection algorithms [RFC3517][RFC4653][RFC5827][RFC5681][RFC6675][RFC7765][FACK][THIN-STREAM][TLP], we've found that together they do not perform well. The main reason is that many of them are based on the classic rule of counting duplicate acknowledgments [RFC5681]. They can either detect loss quickly or accurately, but not both, especially when the sender is application-limited or under reordering that is unpredictable. And under these conditions none of them can detect lost retransmissions well.

Also, these algorithms, including RFCs, rarely address the interactions with other algorithms. For example, FACK may consider a packet is lost while RFC3517 may not. Implementing N algorithms while dealing with N^2 interactions is a daunting task and error-prone.

The goal of RACK is to solve all the problems above by replacing many of the loss detection algorithms above with one simpler, and also more effective, algorithm.

2. Overview

The main idea behind RACK is that if a packet has been delivered out of order, then the packets sent chronologically before that were either lost or reordered. This concept is not fundamentally different from [RFC5681][RFC3517][FACK]. But the key innovation in RACK is to use a per-packet transmission timestamp and widely deployed SACK options to conduct time-based inferences instead of inferring losses with packet or sequence counting approaches.

Using a threshold for counting duplicate acknowledgments (i.e., dupthresh) is no longer reliable because of today's prevalent reordering patterns. A common type of reordering is that the last "runt" packet of a window's worth of packet bursts gets delivered first, then the rest arrive shortly after in order. To handle this effectively, a sender would need to constantly adjust the dupthresh to the burst size; but this would risk increasing the frequency of RTOs on real losses.

Today's prevalent lost retransmissions also cause problems with packet-counting approaches [RFC5681][RFC3517][FACK], since those approaches depend on reasoning in sequence number space. Retransmissions break the direct correspondence between ordering in sequence space and ordering in time. So when retransmissions are lost, sequence-based approaches are often unable to infer and quickly repair losses that can be deduced with time-based approaches.

Instead of counting packets, RACK uses the most recently delivered packet's transmission time to judge if some packets sent previous to that time have "expired" by passing a certain reordering settling window. On each ACK, RACK marks any already-expired packets lost, and for any packets that have not yet expired it waits until the reordering window passes and then marks those lost as well. In either case, RACK can repair the loss without waiting for a (long) RTO. RACK can be applied to both fast recovery and timeout recovery, and can detect losses on both originally transmitted and retransmitted packets, making it a great all-weather recovery mechanism.

3. Requirements

The reader is expected to be familiar with the definitions given in the TCP congestion control [RFC5681] and selective acknowledgment

[RFC2018] RFCs. Familiarity with the conservative SACK-based recovery for TCP [RFC6675] is not expected but helps.

RACK has three requirements:

1. The connection MUST use selective acknowledgment (SACK) options [RFC2018].

2. For each packet sent, the sender MUST store its most recent transmission time with (at least) millisecond granularity. For round-trip times lower than a millisecond (e.g., intra-datacenter communications) microsecond granularity would significantly help the detection latency but is not required.

3. For each packet sent, the sender MUST store whether the packet has been retransmitted or not.

We assume that requirement 1 implies the sender keeps a SACK scoreboard, which is a data structure to store selective acknowledgment information on a per-connection basis. For the ease of explaining the algorithm, we use a pseudo-scoreboard that manages the data in sequence number ranges. But the specifics of the data structure are left to the implementor.

RACK does not need any change on the receiver.

4. Definitions of variables

A sender needs to store these new RACK variables:

"Packet.xmit_ts" is the time of the last transmission of a data packet, including any retransmissions, if any. The sender needs to record the transmission time for each packet sent and not yet acknowledged. The time MUST be stored at millisecond granularity or finer.

"RACK.xmit_ts" is the most recent Packet.xmit_ts among all the packets that were delivered (either cumulatively acknowledged or selectively acknowledged) on the connection.

"RACK.end_seq" is the ending TCP sequence number of the packet that was used to record the RACK.xmit_ts above.

"RACK.RTT" is the associated RTT measured when RACK.xmit_ts, above, was changed. It is the RTT of the most recently transmitted packet that has been delivered (either cumulatively acknowledged or selectively acknowledged) on the connection.

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

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

Note that the Packet.xmit_ts variable is per packet in flight. The RACK.xmit_ts, RACK.RTT, RACK.reo_wnd, and RACK.min_RTT variables are per connection.

5. Algorithm Details

5.1. Transmitting a data packet

Upon transmitting a new packet or retransmitting an old packet, record the time in Packet.xmit_ts. RACK does not care if the retransmission is triggered by an ACK, new application data, an RTO, or any other means.

5.2. Upon receiving an ACK

Step 1: Update RACK.min_RTT.

Use the RTT measurements obtained in [RFC6298] or [RFC7323] to update the estimated minimum RTT in RACK.min_RTT. The sender can track a simple global minimum of all RTT measurements from the connection, or a windowed min-filtered value of recent RTT measurements. This document does not specify an exact approach.

Step 2: Update RACK.reo_wnd.

To handle the prevalent small degree of reordering, RACK.reo_wnd serves as an allowance for settling time before marking a packet lost. By default it is 1 millisecond. We RECOMMEND implementing the reordering detection in [REORDER-DETECT][RFC4737] to dynamically adjust the reordering window. When the sender detects packet reordering RACK.reo_wnd MAY be changed to RACK.min_RTT/4. We discuss more about the reordering window in the next section.

Step 3: Advance RACK.xmit_ts and update RACK.RTT and RACK.end_seq

Given the information provided in an ACK, each packet cumulatively ACKed or SACKed is marked as delivered in the scoreboard. Among all the packets newly ACKed or SACKed in the connection, record the most recent Packet.xmit_ts in RACK.xmit_ts if it is ahead of RACK.xmit_ts. Ignore the packet if any of its TCP sequences has been retransmitted before and either of two condition 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 packet.

2. The packet was last retransmitted less than RACK.min_rtt ago. While it is still possible the packet is spuriously retransmitted because of a recent RTT decrease, we believe that our experience suggests this is a reasonable heuristic.

If this ACK causes a change to RACK.xmit_ts then record the RTT and sequence implied by this ACK:

```
RACK.RTT = Now() - RACK.xmit_ts
RACK.end_seq = Packet.end_seq
```

Exit here and omit the following steps if RACK.xmit_ts has not changed.

Step 4: Detect losses.

For each packet that has not been fully SACKed, if RACK.xmit_ts is after Packet.xmit_ts + RACK.reo_wnd, then mark the packet (or its corresponding sequence range) lost in the scoreboard. The rationale is that if another packet that was sent later has been delivered, and the reordering window or "reordering settling time" has already passed, the packet was likely lost.

If a packet that was sent later has been delivered, but the reordering window has not passed, then it is not yet safe to deem the given packet lost. Using the basic algorithm above, the sender would wait for the next ACK to further advance RACK.xmit_ts; but this risks a timeout (RTO) if no more ACKs come back (e.g, due to losses or application limit). For timely loss detection, the sender MAY install a "reordering settling" timer set to fire at the earliest moment at which it is safe to conclude that some packet is lost. The earliest moment is the time it takes to expire the reordering window of the earliest unacked packet in flight.

This timer expiration value can be derived as follows. As a starting point, we consider that the reordering window has passed if the RACK packet was sent sufficiently after the packet in question, or a sufficient time has elapsed since the RACK packet was S/ACKed, or some combination of the two. More precisely, RACK marks a packet as lost if the reordering window for a packet has elapsed through the sum of:

1. delta in transmit time between a packet and the RACK packet

2. delta in time between the S/ACK of the RACK packet (RACK.ack_ts) and now

So we mark a packet as lost if:

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

If we solve this second condition for "now", the moment at which we can declare a packet lost, then we get:

now > Packet.xmit_ts + RACK.reo_wnd + (RACK.ack_ts - RACK.xmit_ts)

Then (RACK.ack_ts - RACK.xmit_ts) is just the RTT of the packet we used to set RACK.xmit_ts, so this reduces to:

now > Packet.xmit_ts + RACK.RTT + RACK.reo_wnd

The following pseudocode implements the algorithm above. When an ACK is received or the RACK timer expires, call RACK_detect_loss(). The algorithm includes an additional optimization to break timestamp ties by using the TCP sequence space. The optimization is particularly useful to detect losses in a timely manner with TCP Segmentation Offload, where multiple packets in one TSO blob have identical timestamps. It is also useful when the timestamp clock granularity is close to or longer than the actual round trip time.

```
 RACK_detect_loss():
 min_timeout = 0

 For each packet, Packet, in the scoreboard:
     If Packet is already SACKed, ACKed,
        or marked lost and not yet retransmitted:
         Skip to the next packet

     If Packet.xmit_ts > RACK.xmit_ts:
         Skip to the next packet
     If Packet.xmit_ts == RACK.xmit_ts AND // Timestamp tie breaker
Packet.end_seq > RACK.end_seq
         Skip to the next packet

     timeout = Packet.xmit_ts + RACK.RTT + RACK.reo_wnd + 1
     If Now() >= timeout
         Mark Packet lost
     Else If (min_timeout == 0) or (timeout is before min_timeout):
         min_timeout = timeout

 If min_timeout != 0
     Arm a timer to call RACK_detect_loss() after min_timeout
```

6. Analysis and Discussion

6.1. Advantages

The biggest advantage of RACK is that every data packet, whether it is an original data transmission or a retransmission, can be used to detect losses of the packets sent prior to it.

Example: tail drop. Consider a sender that transmits a window of three data packets (P1, P2, P3), and P1 and P3 are lost. Suppose the transmission of each packet is at least RACK.reo_wnd (1 millisecond by default) after the transmission of the previous packet. 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 any timer. 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 packet or sequence count.

Example: lost retransmit. Consider a window of three data packets (P1, P2, P3) that are sent; P1 and P2 are dropped. Suppose the transmission of each packet is at least RACK.reo_wnd (1 millisecond by default) after the transmission of the previous packet. 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 (as a tail drop) but R2 is SACKed; RACK will mark R1 lost for 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 is very common when TCP is being rate-limited, particularly by token bucket policers with large bucket depth and low rate limit. Retransmissions are often lost repeatedly because standard congestion control requires multiple round trips to reduce the rate below the policed rate.

Example: (small) degree of reordering. Consider a common reordering event: a window of packets are sent as (P1, P2, P3). P1 and P2 carry a full payload of MSS octets, but P3 has only a 1-octet payload due to application-limited behavior. Suppose the sender has detected reordering previously (e.g., by implementing the algorithm in [REORDER-DETECT]) 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 falsely mark P1 and P2 lost. We discuss how to reduce the false positives in the end of this section.

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

For some implementations (e.g., Linux), RACK works quite efficiently with TCP Segmentation Offload (TSO). RACK always marks the entire TSO blob lost because the packets in the same TSO blob have the same transmission timestamp. By contrast, the counting based algorithms (e.g., [RFC3517][RFC5681]) may mark only a subset of packets in the TSO blob lost, forcing the stack to perform expensive fragmentation of the TSO blob, or to selectively tag individual packets lost in the scoreboard.

6.2. Disadvantages

RACK requires the sender to record the transmission time of each packet sent at a clock granularity of one millisecond or finer. TCP implementations that record this already for RTT estimation do not require any new per-packet state. But implementations that are not yet recording packet transmission times will need to add per-packet internal state (commonly either 4 or 8 octets per packet) to track transmission times. In contrast, the conventional approach requires one variable to track number of duplicate ACK threshold.

6.3. Adjusting the reordering window

RACK uses a reordering window of min_rtt / 4. It uses the minimum RTT to accommodate reordering introduced by packets traversing slightly different paths (e.g., router-based parallelism schemes) or out-of-order deliveries in the lower link layer (e.g., wireless links using link-layer retransmission). Alternatively, RACK can use the smoothed RTT used in RTT estimation [RFC6298]. However, smoothed RTT can be significantly inflated by orders of magnitude due to congestion and buffer-bloat, which would result in an overly conservative reordering window and slow loss detection. Furthermore, RACK uses a quarter of minimum RTT because Linux TCP uses the same factor in its implementation to delay Early Retransmit [RFC5827] to reduce spurious loss detections in the presence of reordering, and experience shows that this seems to work reasonably well.

One potential improvement is to further adapt the reordering window by measuring the degree of reordering in time, instead of packet distances. But that requires storing the delivery timestamp of each

packet. Some scoreboard implementations currently merge SACKed packets together to support TSO (TCP Segmentation Offload) for faster scoreboard indexing. Supporting per-packet delivery timestamps is difficult in such implementations. However, we acknowledge that the current metric can be improved by further research.

6.4. Relationships with other loss recovery algorithms

The primary motivation of RACK is to ultimately provide a simple and general replacement for some of the standard loss recovery algorithms [RFC5681][RFC6675][RFC5827][RFC4653] and nonstandard ones [FACK][THIN-STREAM]. While RACK can be a supplemental loss detection on top of these algorithms, this is not necessary, because the RACK implicitly subsumes most of them.

[RFC5827][RFC4653][THIN-STREAM] dynamically adjusts the duplicate ACK threshold based on the current or previous flight sizes. RACK takes a different approach, by using only one ACK event and a reordering window. RACK can be seen as an extended Early Retransmit [RFC5827] without a FlightSize limit but with an additional reordering window. [FACK] considers an original packet to be lost when its sequence range is sufficiently far below the highest SACKed sequence. In some sense RACK 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.

Nevertheless RACK is still an experimental algorithm. Since the oldest loss detection algorithm, the 3 duplicate ACK threshold [RFC5681], has been standardized and widely deployed, we RECOMMEND TCP implementations use both RACK and the algorithm specified in Section 3.2 in [RFC5681] for compatibility.

RACK is compatible with and does not interfere with the the standard RTO [RFC6298], RTO-restart [RFC7765], F-RTO [RFC5682] and Eifel algorithms [RFC3522]. This is because RACK only detects loss by using ACK events. It neither changes the timer calculation nor detects spurious timeouts.

Furthermore, RACK naturally works well with Tail Loss Probe [TLP] because a tail loss probe solicit seither an ACK or SACK, which can be used by RACK to detect more losses. RACK can be used to relax TLP’s requirement for using FACK and retransmitting the the highest-sequenced packet, because RACK is agnostic to packet sequence numbers, and uses transmission time instead. Thus TLP can be modified to retransmit the first unacknowledged packet, which can improve application latency.

6.5. Interaction with congestion control

RACK intentionally decouples loss detection from congestion control. RACK only detects losses; it does not modify the congestion control algorithm [RFC5681][RFC6937]. However, RACK may detect losses earlier or later than the conventional duplicate ACK threshold approach does. A packet marked lost by RACK SHOULD NOT be retransmitted until congestion control deems this appropriate (e.g. using [RFC6937]).

RACK is applicable for both fast recovery and recovery after a retransmission timeout (RTO) in [RFC5681]. The distinction between fast recovery or RTO recovery is not necessary because RACK is purely based on the transmission time order of packets. When a packet retransmitted by RTO is acknowledged, RACK will mark any unacked packet sent sufficiently prior to the RTO as lost, because at least one RTT has elapsed since these packets were sent.

6.6. RACK for other transport protocols

RACK can be implemented in other transport protocols. The algorithm can skip step 3 and simplify if the protocol can support unique transmission or packet identifier (e.g. TCP echo options). For example, the QUIC protocol implements RACK [QUIC-LR] .

7. Security Considerations

RACK does not change the risk profile for TCP.

An interesting scenario is ACK-splitting attacks [SCWA99]: for an MSS-size packet sent, the receiver or the attacker might send MSS ACKs that SACK or acknowledge one additional byte per ACK. This would not fool RACK. RACK.xmit_ts would not advance because all the sequences of the packet are transmitted at the same time (carry the same transmission timestamp). In other words, SACKing only one byte of a packet or SACKing the packet in entirety have the same effect on RACK.

8. IANA Considerations

This document makes no request of IANA.

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

9. Acknowledgments

The authors thank Matt Mathis for his insights in FACK and Michael Welzl for his per-packet timer idea that inspired this work. Nandita Dukkipati, Eric Dumazet, Randy Stewart, Van Jacobson, Ian Swett, and Jana Iyengar contributed to the algorithm and the implementations in Linux, FreeBSD and QUIC.

10. References

10.1. Normative References

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

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

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

[RFC4737] Morton, A., Ciavattone, L., Ramachandran, G., Shalunov, S., and J. Perser, "Packet Reordering Metrics", RFC 4737, November 2006.

[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.

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

[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.

[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.

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

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

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

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

10.2. Informative References

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

[TLP] Dukkipati, N., Cardwell, N., Cheng, Y., and M. Mathis, "Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Drops", draft-dukkipati-tcpm-tcp-loss-probe-01 (work in progress), August 2013.

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

[REORDER-DETECT] Zimmermann, A., Schulte, L., Wolff, C., and A. Hannemann, "Detection and Quantification of Packet Reordering with TCP", draft-zimmermann-tcpm-reordering-detection-02 (work in progress), November 2014.

[QUIC-LR] Iyengar, J. and I. Swett, "QUIC Loss Recovery And Congestion Control", draft-tsvwg-quic-loss-recovery-01 (work in progress), June 2016.

[THIN-STREAM] Petlund, A., Evensen, K., Griwodz, C., and P. Halvorsen, "TCP enhancements for interactive thin-stream applications", NOSSDAV , 2008.

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

[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.

Authors' Addresses

Yuchung Cheng
Google, Inc
1600 Amphitheater Parkway
Mountain View, California 94043
USA

Email: ycheng@google.com

Neal Cardwell
Google, Inc
76 Ninth Avenue
New York, NY 10011
USA

Email: ncardwell@google.com

CoRE Working Group
Internet-Draft
Intended status: Best Current Practice
Expires: December 12, 2016

C. Gomez
UPC/i2CAT
J. Crowcroft
University of Cambridge

June 10, 2016

TCP over Constrained-Node Networks
draft-gomez-core-tcp-constrained-node-networks-00

Abstract

This document provides a profile for the Transmission Control Protocol (TCP) over Constrained-Node Networks (CNNs). The overarching goal is to offer simple measures to allow for lightweight TCP implementation and suitable operation in such environments.

Status of This Memo

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

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

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

This Internet-Draft will expire on December 12, 2016.

Table of Contents

1. Introduction

The Internet Protocol suite is being used for connecting Constrained-Node Networks (CNNs) to the Internet, enabling the so-called Internet of Things (IoT) [RFC7228]. In order to meet the requirements that stem from CNNs, the IETF has produced a suite of protocols specifically designed for such environments [I-D.ietf-lwig-energy-efficient].

At the application layer, the Constrained Application Protocol (CoAP) was developed over UDP [RFC7252]. However, the integration of some CoAP deployments with existing infrastructure is being challenged by middleboxes such as firewalls, which may limit UDP-based communications. This is one of the main reasons why a CoAP over TCP specification is being developed [I-D.tschofenig-core-coap-tcp-tls].

On the other hand, other application layer protocols not specifically designed for CNNs are also being considered for the IoT space. Some examples include HTTP/2 and even HTTP/1.1, both of which run over TCP by default [RFC7540][RFC2616]. TCP is also used by non-IETF application-layer protocols in the IoT space such as MQTT and its lightweight variants [MQTTS].

This document provides a profile for TCP over CNNs. The overarching goal is to offer simple measures to allow for lightweight TCP implementation and suitable operation in such environments.

1.1. 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]

2. Characteristics of CNNs relevant for TCP

Constrained nodes are characterized by significant limitations on processing, memory, and energy resources [RFC7228]. The first two dimensions pose constraints on the complexity and on the memory footprint of the protocols that constrained nodes can support. The latter requires techniques to save energy, such as radio duty-cycling in wireless devices [I-D.ietf-lwig-energy-efficient], as well as minimization of the number of messages transmitted/received (and their size).

Constrained nodes often use physical/link layer technologies that have been characterized as 'lossy'. Many such technologies are wireless, therefore exhibiting a relatively high bit error rate. However, some wired technologies used in the CNN space are also lossy (e.g. Power Line Communication).

Some CNNs follow the star topology, whereby one or several hosts are linked to a central device that acts as a router connecting the CNN to the Internet. CNNs may also follow the multihop topology [RFC6606].

3. TCP over CNNs

3.1. Maximum Segment Size (MSS)

Some link layer technologies in the CNN space are characterized by a short data unit payload size, e.g. up to a few tens or hundreds of bytes. For example, the maximum frame size in IEEE 802.15.4 is 127 bytes.

6LoWPAN defined an adaptation layer to support IPv6 over IEEE 802.15.4 networks. The adaptation layer includes a fragmentation mechanism, since IPv6 requires the layer below to support an MTU of 1280 bytes [RFC2460], while IEEE 802.15.4 lacked fragmentation mechanisms. 6LoWPAN defines an IEEE 802.15.4 link MTU of 1280 bytes [RFC4944]. Other technologies, such as Bluetooth LE [RFC7668], ITU-T G.9959 [RFC7428] or DECT-ULE [I-D.ietf-6lo-dect-ule], do support link layer fragmentation. By exploiting this functionality, the adaptation layers to enable IPv6 over such technologies also support an MTU of 1280 bytes.

In order to avoid IP layer fragmentation, the TCP MSS MUST NOT be set to a value greater than 1220 bytes in CNNs. (Note: IP version 6 is assumed.) In any case, the TCP MSS MUST NOT be set to a value leading to an IPv6 datagram size exceeding 1280 bytes.

3.2. Window Size

As per this document, the TCP window size MUST have a size of one segment. This value is appropriate for simple message exchanges in the CNN space, reduces implementation complexity and memory requirements, and reduces overhead (see section 3.6).

A TCP window size of one segment follows the same rationale as the default setting for NSTART in [RFC7252], leading to equivalent operation when CoAP is used over TCP.

3.3. RTO estimation

Traditionally, TCP has used the well known RTO estimation algorithm defined in [RFC6298]. However, experimental studies have shown that another algorithm such as the RTO estimator defined in [I-D.bormann-core-cocoa] (hereinafter, CoCoA RTO) outperforms state-of-art algorithms designed as improvements to RFC 6298 for TCP, in terms of packet delivery ratio, settling time after a burst of messages, and fairness (the latter is specially relevant in multihop networks connected to the Internet through a single device, such as a 6LoWPAN Border Router (6LBR) configured as a RPL root) [Commag]. In fact, CoCoA RTO has been designed specifically considering the challenges of CNNs, in contrast with the RFC 6298 RTO. Therefore, as per this document, CoCoA RTO SHOULD be used in TCP over CNNs. Alternatively, implementors MAY choose the RTO estimation algorithm defined in RFC 6298. One of the two RTO algorithms MUST be implemented.

3.4. Keep-alive and TCP connection lifetime

In CNNs, a TCP connection SHOULD be kept open as long as the two TCP endpoints have more data to exchange or it is envisaged that further segment exchanges will take place within an interval of two hours since the last segment has been sent. A greater interval MAY be used in scenarios where applications exchange data infrequently.

TCP keep-alive messages [RFC1122] MAY be supported by a server, to check whether a TCP connection is active, in order to release state of inactive connections. This may be useful for servers running on memory-constrained devices.

Since the keep-alive timer may not be set to a value lower than two hours [RFC1122], TCP keep-alive messages are not useful to guarantee that filter state records in middleboxes such as firewalls will not be deleted after an inactivity interval typically in the order of a few minutes [RFC6092]. In scenarios where such middleboxes are present, alternative measures to avoid early deletion of filter state records (which might lead to frequent establishment of new TCP connections between the two involved endpoints) include increasing the initial value for the filter state inactivity timers (if possible), and using application layer heartbeat messages.

3.5. Explicit congestion notification

Explicit Congestion Notification (ECN) [RFC3168] MAY be used in CNNs. ECN allows a router to signal in the IP header of a packet that congestion is arising, for example when queue size reaches a certain threshold. If such a packet encapsulates a TCP data packet, an ECN-enabled TCP receiver will echo back the congestion signal to the TCP sender by setting a flag in its next TCP ACK. The sender triggers congestion control measures as if a packet loss had happened. In that case, when the congestion window of a TCP sender has a size of one segment, the TCP sender resets the retransmit timer, and will only be able to send a new packet when the retransmit timer expires [RFC3168]. Effectively, the TCP sender reduces at that moment its sending rate from 1 segment per RTT to 1 segment per default RTO.

ECN can reduce packet losses, since congestion control measures can be applied earlier than after the reception of three duplicate ACKs (if the TCP sender window is large enough, which will not happen as per section 3.2 of this document) or upon TCP sender RTO expiration [RFC2884]. Therefore, the number of retries decreases, which is particularly beneficial in CNNs, where energy and bandwidth resources are typically limited. Furthermore, latency and jitter are also reduced.

ECN is also appropriate in CNNs, since in these environments transactional type interactions are a dominant traffic pattern. Exploiting other possible congestion signals such as the reception of three duplicate ACKs would require the use of greater TCP window sizes than the one specified in this document.

3.6. TCP options

Because this specification mandates a TCP window size of one segment, the following TCP options MUST NOT be supported in CNNs: Window scale [RFC1323], TCP Timestamps [RFC1323], and Selective Acknowledgements (SACK) [RFC2018]. Other TCP options SHOULD NOT be used, in keeping with the principle of lightweight operation.

3.7. Explicit loss notifications

There has been a significant body of research on solutions capable of explicitly indicating whether a TCP segment loss is due to corruption, in order to avoid activation of congestion control mechanisms [ETEN] [RFC2757]. While such solutions may provide significant improvement, they have not been widely deployed and remain as experimental work. In fact, as of today, the IETF has not standardized any such solution.

4. Security Considerations

TBD

5. Acknowledgments

Carles Gomez has been funded in part by the Spanish Government (Ministerio de Educacion, Cultura y Deporte) through the Jose Castillejo grant CAS15/00336. His contribution to this work has been carried out during his stay as a visiting scholar at the Computer Laboratory of the University of Cambridge.

6. References

6.1. Normative References

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

[RFC1323] Jacobson, V., Braden, R., and D. Borman, "TCP Extensions for High Performance", RFC 1323, DOI 10.17487/RFC1323, May 1992, <http://www.rfc-editor.org/info/rfc1323>.

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

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

[RFC2460] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, DOI 10.17487/RFC2460, December 1998, <http://www.rfc-editor.org/info/rfc2460>.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, DOI 10.17487/RFC2616, June 1999, <http://www.rfc-editor.org/info/rfc2616>.

[RFC2757] Montenegro, G., Dawkins, S., Kojo, M., Magret, V., and N. Vaidya, "Long Thin Networks", RFC 2757, DOI 10.17487/RFC2757, January 2000, <http://www.rfc-editor.org/info/rfc2757>.

[RFC2884] Hadi Salim, J. and U. Ahmed, "Performance Evaluation of Explicit Congestion Notification (ECN) in IP Networks", RFC 2884, DOI 10.17487/RFC2884, July 2000, <http://www.rfc-editor.org/info/rfc2884>.

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

[RFC4944] Montenegro, G., Kushalnagar, N., Hui, J., and D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks", RFC 4944, DOI 10.17487/RFC4944, September 2007, <http://www.rfc-editor.org/info/rfc4944>.

[RFC6092] Woodyatt, J., Ed., "Recommended Simple Security Capabilities in Customer Premises Equipment (CPE) for Providing Residential IPv6 Internet Service", RFC 6092, DOI 10.17487/RFC6092, January 2011, <http://www.rfc-editor.org/info/rfc6092>.

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

[RFC6606] Kim, E., Kaspar, D., Gomez, C., and C. Bormann, "Problem Statement and Requirements for IPv6 over Low-Power Wireless Personal Area Network (6LoWPAN) Routing", RFC 6606, DOI 10.17487/RFC6606, May 2012, <http://www.rfc-editor.org/info/rfc6606>.

[RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for Constrained-Node Networks", RFC 7228, DOI 10.17487/RFC7228, May 2014, <http://www.rfc-editor.org/info/rfc7228>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <http://www.rfc-editor.org/info/rfc7252>.

[RFC7428] Brandt, A. and J. Buron, "Transmission of IPv6 Packets over ITU-T G.9959 Networks", RFC 7428, DOI 10.17487/RFC7428, February 2015, <http://www.rfc-editor.org/info/rfc7428>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <http://www.rfc-editor.org/info/rfc7540>.

[RFC7668] Nieminen, J., Savolainen, T., Isomaki, M., Patil, B., Shelby, Z., and C. Gomez, "IPv6 over BLUETOOTH(R) Low Energy", RFC 7668, DOI 10.17487/RFC7668, October 2015, <http://www.rfc-editor.org/info/rfc7668>.

6.2. Informative References

[Commag] A. Betzler, C. Gomez, I. Demirkol, J. Paradells, "CoAP Congestion Control for the Internet of Things", IEEE Communications Magazine, June 2016.

[ETEN] R. Krishnan et al, "Explicit transport error notification (ETEN) for error-prone wireless and satellite networks", Computer Networks 2004.

[I-D.bormann-core-cocoa] Bormann, C., Betzler, A., Gomez, C., and I. Demirkol, "CoAP Simple Congestion Control/Advanced", draft-bormann-core-cocoa-03 (work in progress), October 2015.

[I-D.ietf-6lo-dect-ule] Mariager, P., Petersen, J., Shelby, Z., Logt, M., and D. Barthel, "Transmission of IPv6 Packets over DECT Ultra Low Energy", draft-ietf-6lo-dect-ule-05 (work in progress), May 2016.

[I-D.ietf-lwig-energy-efficient] Gomez, C., Kovatsch, M., Tian, H., and Z. Cao, "Energy-Efficient Features of Internet of Things Protocols", draft-ietf-lwig-energy-efficient-04 (work in progress), February 2016.

[I-D.tschofenig-core-coap-tcp-tls]
Bormann, C., Lemay, S., Technologies, Z., and H. Tschofenig, "A TCP and TLS Transport for the Constrained Application Protocol (CoAP)", draft-tschofenig-core-coap-tcp-tls-05 (work in progress), November 2015.

[MQTTS] U. Hunkeler, H.-L. Truong, A. Stanford-Clark, "MQTT-S: A Publish/Subscribe Protocol For Wireless Sensor Networks", 2008.

Authors' Addresses

Carles Gomez
UPC/i2CAT
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

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.

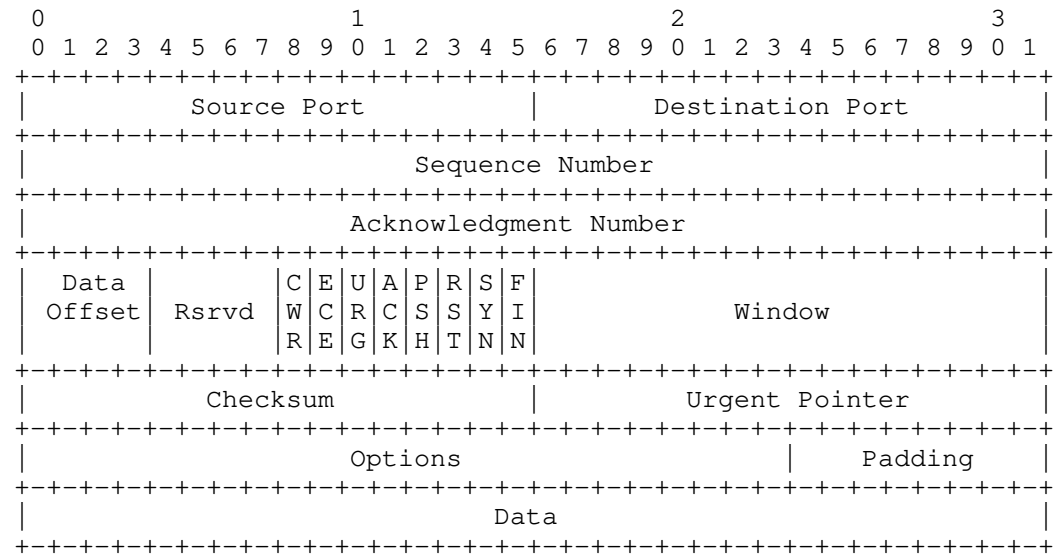
TCP uses port numbers to identify application services and to multiplex distinct flows between hosts.

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.

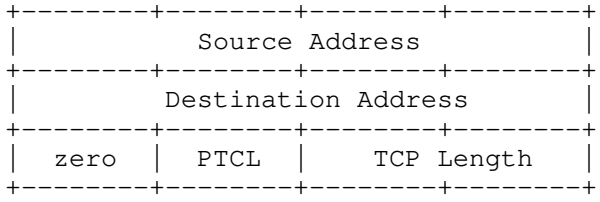


Figure 2: IPv4 Pseudo Header

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).

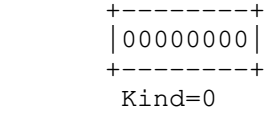
A TCP implementation MUST (MUST-6) ignore without error any TCP option it does not implement, assuming that the option has a length

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].

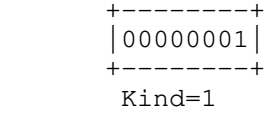
Specific Option Definitions

End of Option List



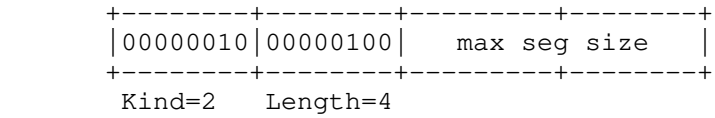
This option code indicates the end of the option list. This might not coincide with the end of the TCP header according to the Data Offset field. This is used at the end of all options, not the end of each option, and need only be used if the end of the options would not otherwise coincide with the end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

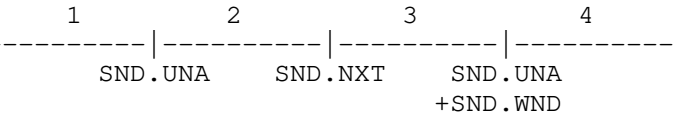
Send Sequence Variables:

```
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
```

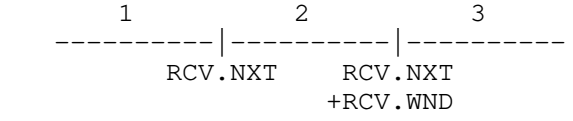
The following diagrams may help to relate some of these variables to the sequence space.



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.

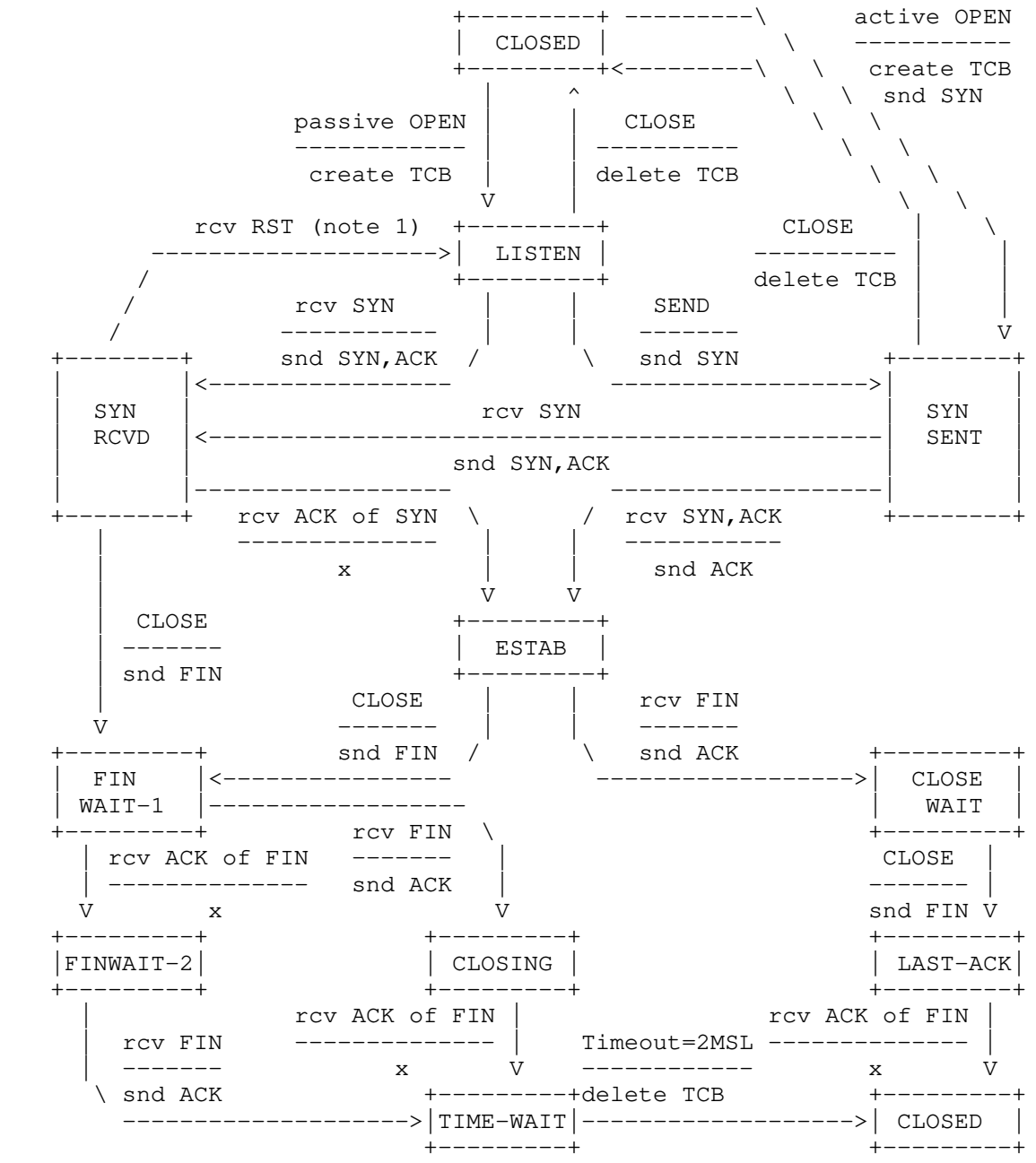


Figure 5: TCP Connection State Diagram

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

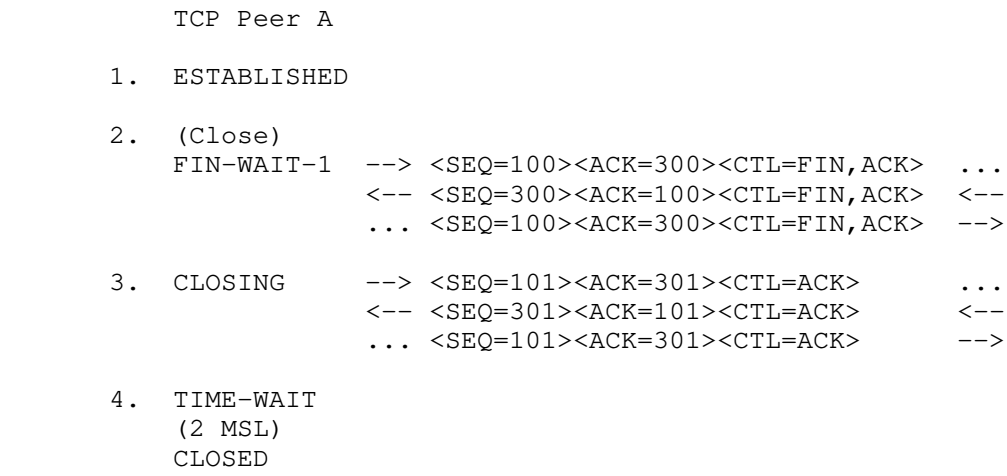


Figure 13: Simultaneous Close Sequence

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

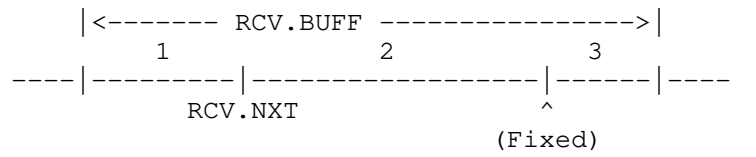
A TCP implementation MUST include a SWS avoidance algorithm in the receiver (MUST-39).

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

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

M. Allman
ICSI

July 27, 2020

Requirements for Time-Based Loss Detection

Status of this Memo

Copyright Notice

Abstract

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

provides high-level requirements for time-based loss detectors appropriate for general use in unicast communication across the Internet. Within the requirements, implementations have latitude to define particulars that best address each situation.

Terminology

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

1 Introduction

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

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

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

(1) Ensuring reliable data delivery.

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

(2) Congestion control.

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

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

Various mechanisms are used to detect losses in a packet stream. Often we use continuous or periodic acknowledgments from the recipient to inform the sender's notion of which pieces of data are missing. However, despite our best intentions and most robust mechanisms we cannot place ultimate faith in receiving such acknowledgments, but can only truly depend on the passage of time. Therefore, our ultimate backstop to ensuring that we detect all loss is a timeout. That is, the sender sets some expectation for how long to wait for confirmation of delivery for a given piece of data. When this time period passes without delivery confirmation the sender concludes the data was lost in transit.

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

- wait long enough to ensure the detection of loss is correct, and

- minimize the amount of delay we impose on applications (before repairing loss) and the network (before we reduce the congestion).

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

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

2 Context

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

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

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

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

- This document does not update or obsolete any existing RFC. These previous specifications---while generally consistent with the requirements in this document---reflect community consensus and this document does not change that consensus.

- The requirements in this document are meant to provide for network safety and, as such, SHOULD be used by all future time-based loss detection mechanisms.

- The requirements in this document may not be appropriate in all cases and, therefore, deviations and variants may be necessary in the future (hence the "SHOULD" in the last bullet). However, inconsistencies MUST be (a) explained and (b) gather consensus.

3 Scope

The principles we outline in this document are protocol-agnostic and widely applicable. We make the following scope statements about the application of the requirements discussed in Section 4:

(S.1) While there are a bevy of uses for timers in protocols---from rate-based pacing to connection failure detection and beyond---this document is focused only on loss detection.

(S.2) The requirements for time-based loss detection mechanisms in this document are for the primary or "last resort" loss detection mechanism whether the mechanism is the sole loss

repair strategy or works in concert with other mechanisms.

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

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

(S.3) The requirements in this document apply only to endpoint-to-endpoint unicast communication. Reliable multicast (e.g., [RFC5740]) protocols are explicitly outside the scope of this document.

Protocols such as SCTP [RFC4960] and MP-TCP [RFC6182] that communicate in a unicast fashion with multiple specific endpoints can leverage the requirements in this document provided they track state and follow the requirements for each endpoint independently. I.e., if host A communicates with addresses B and C, A needs to use independent time-based loss detector instances for traffic sent to B and C.

(S.4) There are cases where state is shared across connections or flows (e.g., [RFC2140], [RFC3124]). State pertaining to time-based loss detection is often discussed as sharable. These situations raise issues that the simple flow-oriented time-based loss detection mechanism discussed in this document does not consider (e.g., how long to preserve state between connections). Therefore, while the general principles given in Section 4 are likely applicable, sharing time-based loss detection information across flows is outside the scope of this document.

4 Requirements

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

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

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

- Premature loss detection can trigger spurious retransmits that could cause issues when a network is already congested.

- Premature loss detection can needlessly cause congestion control to dramatically lower the sender's allowed transmission rate---especially since the rate is already likely low at this stage of the communication. Recovering from such a rate change can taken a relatively long time.

- Finally, as discussed below, sometimes using time-based loss detection and retransmissions can cause ambiguities in assessing the latency of a network path. Therefore, it is especially important for the first latency sample to be free of ambiguities such that there is a baseline for the remainder of the communication.

The specific constant (1 second) comes from the analysis of Internet RTTs found in Appendix A of [RFC6298].

(2) We now specify four requirements that pertain to setting an expected time interval for delivery confirmation.

Often measuring the time required for delivery confirmation is framed as assessing the "round-trip time (RTT)" of the network path. The RTT is the minimum amount of time required to receive delivery confirmation and also often follows protocol behavior whereby acknowledgments are generated quickly after data arrives. For instance, this is the case for the RTO used by TCP [RFC6298] and SCTP [RFC4960]. However, this is somewhat mis-leading and the expected latency is better framed as the "feedback time" (FT). In other words, the expectation is not always simply a network property, but can include additional time before a sender should reasonably expect a response.

For instance, consider a UDP-based DNS request from a client to a recursive resolver [RFC1035]. When the request can be served from the resolver's cache the FT likely well approximates the network RTT between the client and resolver. However, on a cache miss the resolver will request the needed information from one or more authoritative DNS servers, which will non-trivially increase the FT compared to the network RTT between the client and resolver.

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

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

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

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

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

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

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

Internet measurements show that taking only a single FT sample per TCP connection results in a relatively poorly performing RTO mechanism [AP99], hence this requirement that the FT be sampled continuously throughout the lifetime of communication.

As an example, TCP takes an FT sample roughly once per RTT, or if using the timestamp option [RFC7323] on each acknowledgment arrival. [AP99] shows that both these approaches result in roughly equivalent performance for the RTO estimator.

(c) FT observations MAY be taken from non-data exchanges.

Some protocols use non-data exchanges for various reasons---e.g., keepalives, heartbeats, control messages. To the extent that the latency of these exchanges mirrors data exchange, they can be leveraged to take FT samples within the RTO mechanism. Such samples can help protocols keep their RTO accurate during lulls in data transmission. However, given that these messages may not be subject to the same delays as data transmission, we do not take a general view on whether this is useful or not.

(d) An RTO mechanism MUST NOT use ambiguous FT samples.

Assume two copies of some packet X are transmitted at times t0 and t1 and then at time t2 the sender receives confirmation that X in fact arrived. In some cases, it is not clear which copy of X triggered the confirmation and hence the actual FT is either t2-t1 or t2-t0, but which is a mystery. Therefore, in this situation an implementation MUST NOT use either version of the FT sample and hence not update the RTO (as discussed in [KP87,RFC6298]).

There are cases where two copies of some data are transmitted in a way whereby the sender can tell which is being acknowledged by an incoming ACK. E.g., TCP's timestamp option [RFC7323] allows for packets to be uniquely identified and hence avoid the ambiguity. In such cases there is no ambiguity and the resulting samples can update the RTO.

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

This ensures network safety.

An exception to this rule is if an IETF standardized mechanism determines that a particular loss is due to a non-congestion event (e.g., packet corruption). In such a case a congestion

control action is not required. Additionally, congestion control actions taken based on time-based loss detection could be reversed when a standard mechanism post-facto determines that the cause of the loss was not congestion (e.g., [RFC5682]).

(4) Each time the RTO is used to detect a loss, the value of the RTO MUST be exponentially backed off such that the next firing requires a longer interval. The backoff SHOULD be removed after either (a) the subsequent successful transmission of non-retransmitted data, or (b) an RTO passes without detecting additional losses. The former will generally be quicker. The latter covers cases where loss is detected, but not repaired.

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

This ensures network safety.

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

5 Discussion

We note that research has shown the tension between the responsiveness and correctness of time-based loss detection seems to be a fundamental tradeoff in the context of TCP [AP99]. That is, making the RTO more aggressive (e.g., via changing TCP's exponentially weighted moving average (EWMA) gains, lowering the minimum RTO, etc.) can reduce the time required to detect actual loss. However, at the same time, such aggressiveness leads to more cases of mistakenly declaring packets lost that ultimately arrived at the receiver. Therefore, being as aggressive as the requirements given in the previous section allow in any particular situation may not be the best course of action because detecting loss---even if falsely---carries a requirement to invoke a congestion response which will ultimately reduce the transmission rate.

While the tradeoff between responsiveness and correctness seems fundamental, the tradeoff can be made less relevant if the sender can detect and recover from mistaken loss detection. Several mechanisms have been proposed for this purpose, such as Eifel [RFC3522], F-RTO [RFC5682] and DSACK [RFC2883,RFC3708]. Using such mechanisms may allow a data originator to tip towards being more responsive without incurring (as much of) the attendant costs of mistakenly declaring packets to be lost.

Also, note that, in addition to the experiments discussed in [AP99], the Linux TCP implementation has been using various non-standard RTO mechanisms for many years seemingly without large-scale problems (e.g., using different EWMA gains than specified in [RFC6298]). Further, a number of TCP implementations use a steady-state minimum RTO that is less than the 1 second specified in [RFC6298]. While the implication of these deviations from the standard may be more

spurious retransmits (per [AP99]), we are aware of no large-scale network safety issues caused by this change to the minimum RTO. This informs the guidelines in the last section (e.g., there is no minimum RTO specified).

Finally, we note that while allowing implementations to be more aggressive could in fact increase the number of needless retransmissions, the above requirements fail safe in that they insist on exponential backoff and a transmission rate reduction. Therefore, providing implementers more latitude than they have traditionally been given in IETF specifications of RTO mechanisms does not somehow open the flood gates to aggressive behavior. Since there is a downside to being aggressive, the incentives for proper behavior are retained in the mechanism.

6 Security Considerations

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

7 IANA Considerations

This document has no IANA considerations.

Acknowledgments

This document benefits from years of discussions with Ethan Blanton, Sally Floyd, Jana Iyengar, Shawn Ostermann, Vern Paxson, and the members of the TCPM and TCP-IMPL working groups. Ran Atkinson, Yuchung Cheng, David Black, Stewart Bryant, Martin Duke, Wesley Eddy, Gorry Fairhurst, Rahul Arvind Jadhav, Benjamin Kaduk, Mirja Kuhlewind, Nicolas Kuhn, Jonathan Looney and Michael Scharf provided useful comments on previous versions of this document.

Normative References

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

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

Informative References

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

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

[DCCM13] Dukkipati, N., N. Cardwell, Y. Cheng, M. Mathis, "Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses", Internet-Draft draft-dukkipati-tcpm-tcp-loss-probe-01.txt (work in progress), February 2013.

[IS20] Iyengar, I., I. Swett, "QUIC Loss Detection and Congestion Control", Internet-Draft draft-ietf-quic-recovery-27.txt (work in progress), March 2020.

[Jac88] Jacobson, V., "Congestion Avoidance and Control", ACM SIGCOMM, August 1988.

[KP87] Karn, P. and C. Partridge, "Improving Round-Trip Time Estimates in Reliable Transport Protocols", SIGCOMM 87.

[RFC1034] Mockapetris, P. "Domain Names - Concepts and Facilities", RFC 1034, November 1987.

[RFC1035] Mockapetris, P. "Domain Names - Implementation and Specification", RFC 1035, November 1987.

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

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

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

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

[RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, June 2002.

[RFC3522] Ludwig, R., M. Meyer, "The Eifel Detection Algorithm for TCP", RFC 3522, april 2003.

[RFC3708] Blanton, E., M. Allman, "Using TCP Duplicate Selective Acknowledgement (DSACKs) and Stream Control Transmission Protocol (SCTP) Duplicate Transmission Sequence Numbers (TSNs) to Detect Spurious Retransmissions", RFC 3708, February 2004.

[RFC4960] Stweart, R., "Stream Control Transmission Protocol", RFC 4960, September 2007.

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

[RFC5682] Sarolahti, P., M. Kojo, K. Yamamoto, M. Hata, "Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious

Retransmission Timeouts with TCP", RFC 5682, September 2009.

[RFC5740] Adamson, B., C. Bormann, M. Handley, J. Macker, "NACK-Oriented Reliable Multicast (NORM) Transport Protocol", RFC 5740, November 2009.

[RFC6182] Ford, A., C. Raiciu, M. Handley, S. Barre, J. Iyengar, "Architectural Guidelines for Multipath TCP Development", March 2011, RFC 6182.

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

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

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

Authors' Addresses

Mark Allman
International Computer Science Institute
1947 Center St. Suite 600
Berkeley, CA 94704

EMail: mallman@icir.org
http://www.icir.org/mallman

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: May 3, 2017

N. Khademi
M. Welzl
University of Oslo
G. Armitage
Swinburne University of Technology
G. Fairhurst
University of Aberdeen

October 30, 2016

TCP Alternative Backoff with ECN (ABE)
draft-khademi-tcpm-alternativebackoff-ecn-01

Abstract

This memo updates the TCP sender-side reaction to a congestion notification received via Explicit Congestion Notification (ECN). The updated method reduces FlightSize in Congestion Avoidance by a smaller amount than the TCP reaction to loss. The intention is to achieve good throughput when the queue at the bottleneck is smaller than the bandwidth-delay-product of the connection. This is more likely when an Active Queue Management (AQM) mechanism has used ECN to CE-mark a packet, than when a packet was lost. Future versions of this document will also describe a corresponding method for SCTP.

Status of this Memo

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

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

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

This Internet-Draft will expire on May 3, 2017.

Copyright Notice

Table of Contents

1. Definitions

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

2. Introduction

Complementing [I-D.AQM-ECN-benefits], [I-D.ECN-exp] enables wider ECN deployment by updating rules in [RFC3168] that prohibited certain experiments. Specifically, [I-D.ECN-exp] allows for experiments to specify a congestion control response to a CE-marked packet that differs from the response to a dropped packet. This memo defines such a different congestion control response, called "ABE" (Alternative Backoff with ECN). ABE is thus an Experiment in accordance with [I-D.ECN-exp].

[RFC5681] stipulates that TCP congestion control sets "ssthresh" to max(FlightSize / 2, 2*SMSS) in response to packet loss. This corresponds to a backoff multiplier of 0.5 (halving cwnd and sshthresh after packet loss). Consequently, a standard TCP flow using this reaction needs significant network queue space: it can only fully utilise a bottleneck when the length of the link queue (or the AQM dropping threshold) is at least the bandwidth-delay product (BDP) of the flow.

A backoff multiplier of 0.5 is not the only available strategy. As defined in [I-D.CUBIC], CUBIC multiplies the current cwnd by 0.7 in response to loss (the Linux implementation of CUBIC has used a multiplier of 0.7 since kernel version 2.6.25 released in 2008). Consequently, CUBIC utilises paths well even when the bottleneck queue is shorter than the bandwidth-delay product of the flow. However, in the case of a DropTail (FIFO) queue without AQM, such less-aggressive backoff increases the risk of creating a standing queue [CODEL2012].

The standard TCP backoff behaviour defined in [RFC5681] entails reduced link utilisation in situations with short queues and low statistical multiplexing. This memo proposes a concrete sender-side-only congestion control response that remedies this problem.

Devices implementing AQM are likely to be the dominant (and possibly only) source of ECN CE-marking for packets from ECN-capable senders. AQM mechanisms typically strive to maintain a small average queue length, regardless of the bandwidth-delay product of flows passing through them. Receipt of an ECN CE-mark might therefore reasonably be taken to indicate that a small bottleneck queue exists in the

path, and hence the TCP flow would benefit from using a less aggressive backoff multiplier.

Much of the background to this proposal can be found in [ABE2015]. Using a mix of experiments, theory and simulations with standard NewReno and CUBIC, [ABE2015] recommends enabling ECN and letting individual TCP senders use a larger multiplicative decrease factor as a reaction to the receiver reporting ECN CE-marks from AQM-enabled bottlenecks. Such a change is noted to result in "...significant performance gains in lightly-multiplexed scenarios, without losing the delay-reduction benefits of deploying CoDel or PIE" [I-D.CoDel] [I-D.PIE]. This is achieved when reacting to ECN-Echo in Congestion Avoidance by multiplying cwnd and sstthresh with a value in the range [0.7..0.85].

3. Discussion

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

The classic rule-of-thumb dictates that a transport provides a BDP of bottleneck buffering if a TCP connection wishes to optimise path utilisation. A single TCP connection running through such a bottleneck will have opened cwnd up to 2*BDP by the time packet loss occurs. [RFC5681]’s halving of cwnd and ssthresh pushes the TCP connection back to allowing only a BDP of packets in flight -- just sufficient to maintain 100% utilisation of the network path.

AQM schemes like CoDel [I-D.CoDel] and PIE [I-D.PIE] use congestion notifications to constrain the queuing delays experienced by packets, rather than in response to impending or actual bottleneck buffer exhaustion. With current default delay targets, CoDel and PIE both effectively emulate a shallow buffered bottleneck (section II, [ABE2015]) while allowing short traffic bursts into the queue. This interacts acceptably for TCP connections over low BDP paths, or highly multiplexed scenarios (many concurrent TCP connections). However, it interacts badly with lightly-multiplexed cases (few concurrent connections) over a high BDP path. Conventional TCP backoff in such cases leads to gaps in packet transmission and under-utilisation of the path.

The idea to react differently to loss upon detecting an ECN CE-mark pre-dates [ABE2015]. [ICC2002] also proposed using ECN CE-marks to modify TCP congestion control behaviour, using a larger multiplicative decrease factor in conjunction with a smaller additive increase factor to work with RED-based bottlenecks that were not necessarily configured to emulate a shallow queue.

3.2. Focus on ECN as Defined in RFC3168

Some mechanisms rely on ECN semantics that differ from the definitions in [RFC3168] -- for example, Congestion Exposure (ConEx) [RFC7713] and DCTCP [I-D.ietf-tcpm-dctcp] need more accurate ECN information than the feedback mechanism in [RFC3168] offers (defined in [I-D.ietf-tcpm-accurate-ecn]). Such mechanisms allow a sending rate adjustment more frequent than each RTT. These mechanisms are out of the scope of the current document.

3.3. Discussion: Choice of ABE Multiplier

Alternative Backoff with ECN (ABE) decouples a TCP sender's reaction to loss and ECN CE-marks in Congestion Avoidance. The description respectively uses beta_{loss} and beta_{ecn} to refer to the multiplicative decrease factors applied in response to packet loss, and also in response to a receiver indicating that an ECN CE-mark was received on an ECN-enabled TCP connection (based on the terms used in [ABE2015]). For non-ECN-enabled TCP connections, no ECN CE-marks are received and only beta_{loss} applies.

In other words, in response to detected loss:

FlightSize_(n+1) = FlightSize_n * beta_{loss}

and in response to an indication of a received ECN CE-mark:

FlightSize_(n+1) = FlightSize_n * beta_{ecn}

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

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

The Internet has already been running with at least two different beta_{loss} values for several years: the value in [RFC5681] is 0.5, and Linux CUBIC uses 0.7. ABE proposes no change to beta_{loss} used by any current TCP implementations.

beta_{ecn} depends on how the response of a TCP connection to shallow AQM marking thresholds is optimised. beta_{loss} reflects the

preferred response of each TCP algorithm when faced with exhaustion of buffers (of unknown depth) signalled by packet loss. Consequently, for any given TCP algorithm the choice of beta_{ecn} is likely to be algorithm-specific, rather than a constant multiple of the algorithm's existing beta_{loss}.

A range of experiments (section IV, [ABE2015]) with NewReno and CUBIC over CoDel and PIE in lightly-multiplexed scenarios have explored this choice of parameter. These experiments indicate that CUBIC connections benefit from beta_{ecn} of 0.85 (cf. beta_{loss} = 0.7), and NewReno connections see improvements with beta_{ecn} in the range 0.7 to 0.85 (cf. beta_{loss} = 0.5).

4. Specification

This document RECOMMENDS that experimental deployments multiply the FlightSize by 0.8 and reduce the slow start threshold 'ssthresh' in Congestion Avoidance in response to reception of a TCP segment that sets the ECN-Echo flag.

5. Status of the Update

This update is a sender-side only change. Like other changes to congestion-control algorithms it does not require any change to the TCP receiver or to network devices (except to enable an ECN-marking algorithm [RFC3168] [RFC7567]). If the method is only deployed by some TCP senders, and not by others, the senders that use this method can gain advantage, possibly at the expense of other flows that do not use this updated method. This advantage applies only to ECN-marked packets and not to loss indications. Hence, the new method can not lead to congestion collapse.

The present specification has been assigned an Experimental status, to provide Internet deployment experience before being proposed as a Standards-Track update.

6. Acknowledgements

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

The authors would like to thank the following people for their contributions to [ABE2015]: Chamil Kulatunga, David Ros, Stein

Gjessing, Sebastian Zander. Thanks to (in alphabetical order) Bob Briscoe, Markku Kojo, John Leslie, Dave Taht and the TCPM WG for providing valuable feedback on this document.

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

7. IANA Considerations

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

This memo includes no request to IANA.

8. Implementation Status

ABE is implemented as a patch for Linux and FreeBSD. It is meant for research and available for download from http://heim.ifi.uio.no/naeemk/research/ABE/ This code was used to produce the test results that are reported in [ABE2015].

9. Security Considerations

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

This document describes a change to TCP congestion control with ECN that will typically lead to a change in the capacity achieved when flows share a network bottleneck. Similar unfairness in the way that capacity is shared is also exhibited by other congestion control mechanisms that have been in use in the Internet for many years (e.g., CUBIC [I-D.CUBIC]). Unfairness may also be a result of other factors, including the round trip time experienced by a flow. This advantage applies only to ECN-marked packets and not to loss indications, and will therefore not lead to congestion collapse.

10. Revision Information

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

-01. This I-D now refers to draft-black-tsvwg-ecn-experimentation-02, which replaces draft-khademi-tsvwg-ecn-response-00 to make a broader update to

RFC3168 for the sake of allowing experiments. As a result, some of the motivating and discussing text that was moved from draft-khademi-alternativebackoff-ecn-03 to draft-khademi-tsvwg-ecn-response-00 has now been re-inserted here.

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

11. References

11.1. Normative References

[I-D.ECN-exp] Black, D., "Explicit Congestion Notification (ECN) Experimentation", Internet-draft, IETF work-in-progress draft-black-tsvwg-ecn-experimentation-02, October 2016.

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

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

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

[RFC7567] Baker, F., Ed. and G. Fairhurst, Ed., "IETF Recommendations Regarding Active Queue Management", BCP 197, RFC 7567, DOI 10.17487/RFC7567, July 2015, <http://www.rfc-editor.org/info/rfc7567>.

11.2. Informative References

[ABE2015] Khademi, N., Welzl, M., Armitage, G., Kulatunga, C., Ros, D., Fairhurst, G., Gjessing, S., and S. Zander, "Alternative Backoff: Achieving Low Latency and High Throughput with ECN and AQM", CAIA Technical Report CAIA-TR-150710A, Swinburne University of Technology, July 2015, <http://caia.swin.edu.au/reports/150710A/

CAIA-TR-150710A.pdf>.

[CODEL2012]
Nichols, K. and V. Jacobson, "Controlling Queue Delay", July 2012, <http://queue.acm.org/detail.cfm?id=2209336>.

[I-D.AQM-ECN-benefits]
Fairhurst, G. and M. Welzl, "The Benefits of using Explicit Congestion Notification (ECN)", Internet-draft, IETF work-in-progress draft-ietf-aqm-ecn-benefits-08, November 2015.

[I-D.CUBIC]
Rhee, I., Xu, L., Ha, S., Zimmermann, A., Eggert, L., and R. Scheffenegger, "CUBIC for Fast Long-Distance Networks", Internet-draft, IETF work-in-progress draft-ietf-tcpm-cubic-02, August 2016.

[I-D.CoDel]
Nichols, K., Jacobson, V., McGregor, V., and J. Iyengar, "Controlled Delay Active Queue Management", Internet-draft, IETF work-in-progress draft-ietf-aqm-codel-04, June 2016.

[I-D.PIE] Pan, R., Natarajan, P., Baker, F., and G. White, "PIE: A Lightweight Control Scheme To Address the Bufferbloat Problem", Internet-draft, IETF work-in-progress draft-ietf-aqm-pie-10, September 2016.

[I-D.ietf-tcpm-accurate-ecn]
Briscoe, B., Kuehlewind, M., and R. Scheffenegger, "More Accurate ECN Feedback in TCP", draft-ietf-tcpm-accurate-ecn-01 (work in progress), June 2016.

[I-D.ietf-tcpm-dctcp]
Bensley, S., Eggert, L., Thaler, D., Balasubramanian, P., and G. Judd, "Datacenter TCP (DCTCP): TCP Congestion Control for Datacenters", draft-ietf-tcpm-dctcp-02 (work in progress), July 2016.

[ICC2002] Kwon, M. and S. Fahmy, "TCP Increase/Decrease Behavior with Explicit Congestion Notification (ECN)", IEEE ICC 2002, New York, New York, USA, May 2002, <http://dx.doi.org/10.1109/ICC.2002.997262>.

[RFC7713] Mathis, M. and B. Briscoe, "Congestion Exposure (ConEx) Concepts, Abstract Mechanism, and Requirements", RFC 7713,

DOI 10.17487/RFC7713, December 2015,
<http://www.rfc-editor.org/info/rfc7713>.

Authors' Addresses

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

Email: naeemk@ifi.uio.no

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

Email: michawe@ifi.uio.no

Grenville Armitage
Centre for Advanced Internet Architectures
Swinburne University of Technology
PO Box 218
John Street, Hawthorn
Victoria, 3122
Australia

Email: garmitage@swin.edu.au

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

Email: gorry@erg.abdn.ac.uk

TCPM WG
Internet Draft
Intended status: Informational
Obsoletes: 2140
Expires: July 2019

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

TCP Control Block Interdependence
draft-touch-tcpm-2140bis-06.txt

Status of this Memo

Abstract

This memo updates and replaces RFC 2140's description of interdependent TCP control blocks, in which part of the TCP state is shared among similar concurrent or consecutive connections. TCP state includes a combination of parameters, such as connection state, current round-trip time estimates, congestion control information, and process information. Most of this state is maintained on a per-connection basis in the TCP Control Block (TCB), but implementations can (and do) share certain TCB information across connections to the same host. Such sharing is intended to improve overall transient transport performance, while maintaining backward-compatibility with existing implementations. The sharing described herein is limited to only the TCB initialization and so has no effect on the long-term behavior of TCP after a connection has been established.

Table of Contents

1. Introduction

TCP is a connection-oriented reliable transport protocol layered over IP [RFC793]. Each TCP connection maintains state, usually in a data structure called the TCP Control Block (TCB). The TCB contains information about the connection state, its associated local process, and feedback parameters about the connection's transmission properties. As originally specified and usually implemented, most TCB information is maintained on a per-connection basis. Some implementations can (and now do) share certain TCB information across connections to the same host [RFC2140]. Such sharing is intended to lead to better overall transient performance, especially for numerous short-lived and simultaneous connections, as often used in the World-Wide Web [Be94],[Br02].

This document updates RFC 2140's discussion of TCB state sharing and provides a complete replacement for that document. This state sharing affects only TCB initialization [RFC2140] and thus has no effect on the long-term behavior of TCP after a connection has been established. Path information shared across SYN destination port numbers assumes that TCP segments having the same host-pair experience the same path properties, irrespective of TCP port numbers. The observations about TCB sharing in this document apply similarly to any protocol with congestion state, including SCTP [RFC4960] and DCCP [RFC4340], as well as for individual subflows in Multipath TCP [RFC6824].

2. Conventions used in this document

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

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

In this document, the characters ">>" preceding an indented line(s) indicates a statement using the key words listed above. This convention aids reviewers in quickly identifying or finding the portions of this RFC covered by these keywords.

3. Terminology

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

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

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

4. The TCP Control Block (TCB)

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

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

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

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

5. TCB Interdependence

There are two cases of TCB interdependence. Temporal sharing occurs when the TCB of an earlier (now CLOSED) connection to a host is used to initialize some parameters of a new connection to that same host, i.e., in sequence. Ensemble sharing occurs when a currently active connection to a host is used to initialize another (concurrent) connection to that host.

6. An Example of Temporal Sharing

The TCB data cache is accessed in two ways: it is read to initialize new TCBs and written when more current per-host state is available. New TCBs can be initialized using context from past connections as follows:

TEMPORAL SHARING - TCB Initialization

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

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

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

Sharing sendMSS information affects only data in the SYN of the next connection, because sendMSS information is typically included in most TCP SYN segments. Caching PMTU can accelerate the efficiency of PMTUD, but can also result in black-holing until corrected if in error. Caching MMS_R and MMS_S may be of little direct value as they are reported by the local IP stack anyway.

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

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

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

TEMPORAL SHARING - Option info

Cached	New
old_TFO_Cookie	old_TFO_Cookie
old_TFO_Failure	old_TFO_Failure

TEMPORAL SHARING - Cache Updates

Cached TCB	Current TCB	when?	New Cached TCB
old_MMS_S	curr_ MMS_S	OPEN	curr MMS_S
old_MMS_R	curr_ MMS_R	OPEN	curr_MMS_R
old_sendMSS	curr_sendMSS	MSSopt	curr_sendMSS
old_PMTU	curr_PMTU	PMTUD	curr_PMTU
old_RTT	curr_RTT	CLOSE	merge(curr,old)
old_RTTvar	curr_RTTvar	CLOSE	merge(curr,old)
old_option	curr option	ESTAB	(depends on option)
old_ssthresh	curr_ssthresh	CLOSE	merge(curr,old)
old_snd_cwnd	curr_snd_cwnd	CLOSE	merge(curr,old)

Caching PMTU and sendMSS is trivial; reported values are cached, and the most recent values are used. The cache is updated when the MSS option is received in a SYN or after PMTUD (i.e., when an ICMPv4 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. There is no particular benefit to caching MMS_S and MMS R as these are reported by the local IP stack.

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

RTT values are updated by a more complicated mechanism [RFC1644][Ja86]. Dynamic RTT estimation requires a sequence of RTT measurements. As a result, the cached RTT (and its variance) is an average of its previous value with the contents of the currently active TCB for that host, when a TCB is closed. RTT values are updated only when a connection is closed. The method for merging old

and current values needs to attempt to reduce the transient for new connections.

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

TEMPORAL SHARING - Option info Updates

Cached	Current	when?	New Cached
old_TFO_Cookie	old_TFO_Cookie	ESTAB	old_TFO_Cookie
old_TFO_Failure	old_TFO_Failure	ESTAB	old_TFO_Failure

7. An Example of Ensemble Sharing

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

ENSEMBLE SHARING - TCB Initialization

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

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

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

ENSEMBLE SHARING Option info

Cached	New
old_TFO_Cookie	old_TFO_Cookie
old_TFO_Failure	old_TFO_Failure

ENSEMBLE SHARING - Cache Updates

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

For ensemble sharing, TCB information should be cached as early as possible, sometimes before a connection is closed. Otherwise, opening multiple concurrent connections may not result in TCB data sharing if no connection closes before others open. The amount of work involved in updating the aggregate average should be minimized, but the resulting value should be equivalent to having all values measured within a single connection. The function "rtt_update" in

the ensemble sharing table indicates this operation, which occurs whenever the RTT would have been updated in the individual TCP connection. As a result, the cache contains the shared RTT variables, which no longer need to reside in the TCB [Ja86].

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

ENSEMBLE SHARING - Option info Updates

Cached	Current	when?	New Cached
old_TFO_Cookie	old_TFO_Cookie	ESTAB	old_TFO_Cookie
old_TFO_Failure	old_TFO_Failure	ESTAB	old_TFO_Failure

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

There are several ways to initialize the congestion window in a new TCB among an ensemble of current connections to a host. Current TCP implementations initialize it to four segments as standard [rfc3390] and 10 segments experimentally [RFC6928] and T/TCP hinted that it should be initialized to the old window size [RFC1644]. In the former cases, the assumption is that new connections should behave as conservatively as possible. In the latter T/TCP case, no accommodation is made for concurrent aggregate behavior. The algorithm described in [Ba12] adjusts the initial cwnd depending on the cwnd values of ongoing connections.

8. Compatibility Issues

For the congestion and current window information, the initial values computed by TCB interdependence may not be consistent with the long-term aggregate behavior of a set of concurrent connections between the same endpoints. Under conventional TCP congestion control, if a single existing connection has converged to a congestion window of 40 segments, two newly joining concurrent connections assume initial windows of 10 segments [RFC6928], and the

current connection's window doesn't decrease to accommodate this additional load and connections can mutually interfere. One example of this is seen on low-bandwidth, high-delay links, where concurrent connections supporting Web traffic can collide because their initial windows were too large, even when set at one segment.

The authors of [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.

Due to mechanisms like ECMP and LAG [RFC7424], TCP connections sharing the same host-pair may not always share the same path. This does not matter for host-specific information such as RWIN and TCP option state, such as TFOinfo. When TCB information is shared across different SYN destination ports, path-related information can be incorrect; however, the impact of this error is potentially diminished if (as discussed here) TCB sharing affects only the transient event of a connection start or if TCB information is shared only within connections to the same SYN destination port. In case of Temporal Sharing, TCB information could also become invalid over time. Because this is similar to the case when a connection becomes idle, mechanisms that address idle TCP connections (e.g., [RFC7661]) could also be applied to TCB cache management, especially when TCP Fast Open is used [RFC7413].

There may be additional considerations to the way in which TCB interdependence rebalances congestion feedback among the current connections, e.g., it may be appropriate to consider the impact of a connection being in Fast Recovery [RFC5861] or some other similar unusual feedback state, e.g., as inhibiting or affecting the calculations described herein.

TCP is sometimes used in situations where packets of the same host-pair always take the same path. Because ECMP and LAG examine TCP port numbers, they may not be supported when TCP segments are encapsulated, encrypted, or altered - for example, some Virtual Private Networks (VPNs) are known to use proprietary UDP encapsulation methods. Similarly, they cannot operate when the TCP header is encrypted, e.g., when using IPsec ESP. TCB interdependence among the entire set sharing the same endpoint IP addresses should work without problems under these circumstances. Moreover, measures to increase the probability that connections use the same path could be applied: e.g., the connections could be given the same IPv6 flow label. TCB interdependence can also be extended to sets of host IP

address pairs that share the same network path conditions, such as when a group of addresses is on the same LAN (see Section 9).

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

9. Implications

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

TCB interdependence pushes some of the TCP implementation from the traditional transport layer (in the ISO model), to the network layer. This acknowledges that some state is in fact per-host-pair or can be per-path as indicated solely by that host-pair. Transport protocols typically manage per-application-pair associations (per stream), and network protocols manage per-host-pair and path associations (routing). Round-trip time, MSS, and congestion information could be more appropriately handled in a network-layer fashion, aggregated among concurrent connections, and shared across connection instances [RFC3124].

An earlier version of RTT sharing suggested implementing RTT state at the IP layer, rather than at the TCP layer [Ja86]. Our observations are for sharing state among TCP connections, which avoids some of the difficulties in an IP-layer solution. One such problem is determining the associated prior outgoing packet for an incoming packet, to infer RTT from the exchange. Because RTTs are

still determined inside the TCP layer, this is simpler than at the IP layer. This is a case where information should be computed at the transport layer, but could be shared at the network layer.

Per-host-pair associations are not the limit of these techniques. It is possible that TCBs could be similarly shared between hosts on a subnet or within a cluster, because the predominant path can be subnet-subnet, rather than host-host. Additionally, TCB interdependence can be applied to any protocol with congestion state, including SCTP [RFC4960] and DCCP [RFC4340], as well as for individual subflows in Multipath TCP [RFC6824].

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

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

10. Implementation Observations

The observation that some TCB state is host-pair specific rather than application-pair dependent is not new and is a common engineering decision in layered protocol implementations. A discussion of sharing RTT information among protocols layered over IP, including UDP and TCP, occurred in [Ja86]. Although now deprecated, T/TCP was the first to propose using caches in order to maintain TCB states (see Appendix A for more information).

The table below describes the current implementation status for some TCB information in Linux kernel version 4.6, FreeBSD 10 and Windows (as of October 2016). In the table, "shared" only refers to temporal sharing.

TCB data	Status
old MMS_S	Not shared
old MMS_R	Not shared
old_sendMSS	Cached and shared in Linux (MSS)
old PMTU	Cached and shared in FreeBSD and Windows (PMTU)
old_RTT	Cached and shared in FreeBSD and Linux
old_RTTvar	Cached and shared in FreeBSD
old TFOinfo	Cached and shared in Linux and Windows
old_snd_cwnd	Not shared
old_ssthresh	Cached and shared in FreeBSD and Linux: FreeBSD: arithmetic mean of ssthresh and previous value if a previous value exists; Linux: depending on state, max(cwnd/2, ssthresh) in most cases

11. Updates to RFC 2140

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

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

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

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

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

12. Security Considerations

These presented implementation methods do not have additional ramifications for explicit attacks. They may be susceptible to denial-of-service attacks if not otherwise secured. For example, an application can open a connection and set its window size to zero, denying service to any other subsequent connection between those hosts.

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

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

13. IANA Considerations

There are no IANA implications or requests in this document.

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

14. References

14.1. Normative References

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

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

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

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

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

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

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

14.2. Informative References

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

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

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

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

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

[Ja86] Jacobson, V., (mail to public list "tcp-ip", no archive found), 1986.

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

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

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

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

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

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

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

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

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

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

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

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

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

[RFC5861] Allman, M., Paxson, V., Blanton, E., "TCP Congestion Control," RFC 5861, Sept. 2009.

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

[RFC6824] Ford, A., Raiciu, C., Handley, M., Bonaventure, O., "TCP Extensions for Multipath Operation with Multiple Addresses," RFC 6824, Jan. 2013.

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

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

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

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

15. Acknowledgments

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

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

16. Change log

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

06:

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

05:

- Fixed some TBDs.

04:

- Removed BCP-style recommendations and fixed some TBDs.

03:

- Updated Touch's affiliation and address information

02:

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

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

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

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

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

- Added information about option sharing, listed options in the appendix

Authors' Addresses

Joe Touch

Manhattan Beach, CA 90266
USA

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

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

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

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

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

17. Appendix A: TCB sharing history

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

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

Temporal sharing of cached TCB data was originally implemented in the SunOS 4.1.3 T/TCP extensions [Br94] and the FreeBSD port of same [FreeBSD]. As mentioned before, only the MSS and RTT parameters were cached, as originally specified in [RFC1379]. Later discussion of T/TCP suggested including congestion control parameters in this cache [RFC1644].

18. Appendix B: Options

In addition to the options that can be cached and shared, this memo also lists all options for which state should *not* be kept. This list is meant to avoid work duplication and should be removed upon publication.

Obsolete (MUST NOT keep state):

ECHO

ECHO REPLY

PO Conn permitted

PO service profile

CC

CC.NEW

CC.ECHO

Alt CS req

Alt CS data

No state to keep:

EOL

NOP

WS

SACK

TS

MD5

TCP-AO

EXP1

EXP2

MUST NOT keep state:

Skeeter (DH exchange - might be obsolete, though)

Bubba (DH exchange - might really be obsolete, though)

Trailer CS

SCPS capabilities

S-NACK

Records boundaries

Corruption experienced

SNAP

TCP Compression

Quickstart response

UTO

MPTCP (can we cache when this fails?)

TFO success

MAY keep state:

MSS

TFO failure (so we don't try again, since it's optional)

MUST keep state:

TFP cookie (if TFO succeeded in the past)