Internet Engineering Task Force
Internet-Draft
Intended status: Experimental
Expires: January 21, 2016

A. Agache
C. Raiciu
University Politehnica of Bucharest

July 20, 2015

TCP Sendbuffer Advertising
draft-agache-tcpm-sndbufadv-00

Abstract

Network operators have difficulty in understanding the end-to-end performance of TCP connections through their networks. By observing packets at different vantage points on their path and maintaining per flow state, network operators can detect packet losses, retransmission and estimate RTTs, among other metrics. A key information needed by networks is whether a connection is limited by the network or by the application. This information is very difficult to accurately infer by passive measurements.

We propose to advertise sendbuffer occupancy in TCP: each segment will carry the amount of backlogged data present in the sender's buffer. This information allows networks to discern between application-limited, network-limited and flow-control limited flows, creating new avenues of network optimization.

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 21, 2016.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Requirements Language 2
2. Introduction 2
3. TCP Sendbuffer Structure 3
4. Negotiating sendbuffer advertising 4
5. Encoding sendbuffer information 5
6. References 6
  6.1. Normative References 6
  6.2. Informative References 6
Authors' Addresses 7

1. Requirements Language

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

2. Introduction

Aggregate link statistics, such as packet and loss counts, are easily available in modern networks, but they convey a fairly limited picture of network performance. In many cases, the network needs information about individual flows' demand for bandwidth to take the appropriate resource allocation decisions.

One example is a mobile phone streaming audio or video over a WiFi connection. The default strategy is to always stick to WiFi when available, despite the fact that performance may be terrible and seriously impair user experience. If the mobile network knew the multimedia stream needs more bandwidth, it could fire-up the cellular connection and migrate traffic over there by using mobile client offloading software relying on Multipath TCP [NSDI-12] or Mobile IP [RFC5944].

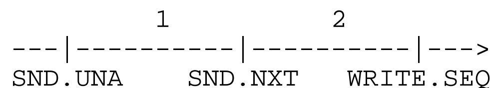
Another example is in datacenters with Clos topologies (such as the popular FatTree topology [FatTree]), where elephant flows are randomly placed on paths with flow-level Equal Cost Multipath

Routing; when one or more elephant flows are placed on the same link, performance degrades despite existing capacity elsewhere in the network. The network can reroute such flows by using tunnels or programmable switches (e.g. Openflow) but the one thing missing is the information regarding which flows could utilize more capacity if given a better path.

Determining if a TCP connection is network limited or not is difficult to do by passive monitoring. The network needs to keep per-flow state, to estimate the sender congestion window and to accurately monitor flight-size. When flight-size is smaller than the congestion window and the receive window, the connection is limited by the application and does not need more capacity.

We propose that each TCP segment should also encode the amount of backlogged data in the TCP sendbuffer. This information enables network boxes and receivers to easily identify connections that need more capacity. Our goal is to have this extension "always on", and it is therefore very important to reduce its overhead. Next, we discuss how to compute and report the amount of backlogged data. We follow with a discussion of signaling options for conveying sendbuffer information.

3. TCP Sendbuffer Structure



1 - sequence numbers of unacknowledged, in flight data
2 - sequence numbers of backlogged data.

Anatomy of the TCP Sendbuffer

The figure above shows the anatomy of the TCP sendbuffer. SND.UNA represents the oldest sequence number sent but not yet acknowledged. At the other end there is WRITE.SEQ, the tail sequence number of data held in the sendbuffer. Somewhere in-between we have SND.NXT, the sequence number of the next byte to be sent. From SND.NXT to WRITE.SEQ we have backlogged data, written by the application but not yet transmitted.

SND.NXT is constrained by both the receive window and the congestion window as follows:

```
SND.NXT <= SND.UNA + min(SND.WND, SND.CWND)
```

As long as the receive window is not a bottleneck, and in the absence of hardware issues or software bugs, having SND.NXT smaller than WRITE.SEQ indicates that the congestion window is not large enough, so the connection is network limited at that point in time. The easiest way to implement sendbuffer advertising is to simply copy the amount of backlogged data (WRITE.SEQ-SND.NXT) into the segment when it leaves the TCP stack. However, this will result in non-zero sendbuffer advertisement when the connection is application-limited but the application writes bursts of a few packets. These packets will be sent out immediately on the wire, yet the first packets in the burst will report that the application is backlogged, when in fact it isn't.

To correctly implement sendbuffer advertisement, the sender MUST advertise the amount of backlogged according to the formula below:

```
SEG.SNDBUF = WRITE.SEQ-SND.UNA - min(SND.WND, SND.CWND),
              if WRITE.SEQ > SND.UNA + min(SND.WND, SND.CWND)

SEG.SNDBUF = 0, otherwise
```

This formula ensures that if an application write fits in the current receive and congestion windows, all the resulting segments will advertise zero backlog data.

4. Negotiating sendbuffer advertising

The standard way to extend TCP is to negotiate the extension during the three-way handshake. The TCP option space, however, is already very crowded in the SYN exchange. Until solutions that extend the TCP option space are standardized, negotiation in the SYN exchange is, in our view, not a feasible option for sendbuffer advertising.

Fortunately, sendbuffer advertising is a sender-side only modification to TCP, and the information it makes available can be used anyone that understands it, be it the network or the receiver. This implies that we can simply bypass the three way handshake as long as the actual encoding of the sendbuffer information in TCP segments does not have negative effects to legacy routers, middleboxes and TCP receivers. We discuss encoding in the next section.

TCP sendbuffer advertising will therefore be a simple sender-only enhancement to the TCP stack that can be enabled by using system-wide configuration (e.g. sysctl in Linux).

5. Encoding sendbuffer information

In this section we discuss two encoding alternatives for sendbuffer information: as new TCP options, in the acknowledgement field of data segments and in the receive-window field.

The first solution is to simply encode sendbuffer information in a new TCP option on every segment carrying data in a TCP connection, without negotiating this extension in the three way handshake. This only adds 6B of overhead to each TCP segment. This option is feasible only when there is sufficient space in the TCP option field of the corresponding data segment.

Avoding the option negotiation will work really well in datacenters where it can be ensured out-of-band that all machines either know sendbuffer advertising or are unaffected by segments carrying new options. In the Internet, before advertising sendbuffer information in new TCP options we need to ensure that: a) existing TCP stacks are robust to unknown options, simply ignoring them, and b) middleboxes do not drop segments carrying unknown options. Existing studies [IMC-11] imply that the wide majority of network paths either allow unknown options or drop the options, allowing the segments through. Only a very small fraction of paths drop the segments with unknown options. To cope with such cases, the implementation MUST NOT include sendbuffer information on retransmitted packets, to ensure that the connection makes some progress even in the presence of such middleboxes.

Our second solution is based on the observation that while TCP itself is bidirectional, most connections in practice will transfer data unidirectionally most times. The endpoints can be either data senders or receivers at different moments, but they rarely act as both at the same time. When traffic is unidirectional, the sender sends the same value for the acknowledgement number and receive window field over and over again.

We propose to reuse one or both of these fields to advertise sendbuffer information instead when traffic is unidirectional. To detect unidirectional traffic, the sender will maintain a state variable called SND.NUM_SEG that is initially set to zero, and is zeroed whenever a segment with a valid ACK field is sent out. SND.NUM_SEG will be incremented whenever a segment is received. A sendbuffer advertisment SHOULD be encoded in outgoing segments only when SND.NUM_SEG = 0.

Sendbuffer advertising will encode the proper value in the ACK field and NOT set the ACK flag. This ensures the receiver and other on-path hosts will ignore the field altogether. We still need, however,

to inform parties interested in sendbuffer information they can use the value of the ACK field.

In datacenters, we can simply define one of the reserved TCP flags as the sendbuffer advertisement flag. When this flag is set, the sendbuffer value is encoded in the ACK field. The sendbuffer advertisement flag and the ACK flag CANNOT be set simultaneously.

In the Internet, redefining the meaning of one of the reserved flags will simply not work through existing middleboxes; additionally, certain middleboxes may zero the ACK field when the ACK flag is not set. In this context, we propose to use the receive window field in segments carrying sendbuffer information to encode a checksum of this information. Interested parties will: a) scan for data segments with the ACK flag not set, b) compute a 1's complement checksum of the ACK field and check it against the receive window field. In case of a match, the sendbuffer information can be used. To understand the feasibility of this encoding, however, tests must to be conducted to check the behaviour of middleboxes when the ACK flag is not set.

6. References

6.1. Normative References

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

6.2. Informative References

[FatTree] Al-Fares, M., Loukissas, A., and A. Vahdat, "A scalable, commodity data center network architecture", 2008, <http://doi.acm.org/10.1145/1402958.1402967>.

[IMC-11] Honda, M., Nishida, Y., Raiciu, C., Greenhalgh, A., Handley, M., and H. Tokuda, "Is it still possible to extend tcp?", 2011, <http://doi.acm.org/10.1145/2068816.2068834>.

[NSDI-12] Raiciu, C., Paasch, C., Barre, S., Ford, A., Honda, M., Duchene, F., Bonaventure, O., and M. Handley, "How hard can it be? designing and implementing a deployable multipath tcp", 2012, <http://dl.acm.org/citation.cfm?id=2228298.2228338>.

[RFC5944] Perkins, C., "IP Mobility Support for IPv4, Revised", RFC 5944, November 2010.

Authors' Addresses

Alexandru Agache
University Politehnica of Bucharest
Splaiul Independentei 313
Bucharest
Romania

Email: alexandru.agache@cs.pub.ro

Costin Raiciu
University Politehnica of Bucharest
Splaiul Independentei 313
Bucharest
Romania

Email: costin.raiciu@cs.pub.ro

Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 8, 2016

S. Bensley
Microsoft
L. Eggert
NetApp
D. Thaler
P. Balasubramanian
Microsoft
G. Judd
Morgan Stanley
July 7, 2015

Microsoft's Datacenter TCP (DCTCP): TCP Congestion Control for Datacenters
draft-bensley-tcpm-dctcp-05

Abstract

This memo describes Datacenter TCP (DCTCP), an improvement to TCP congestion control for datacenter traffic. DCTCP uses improved Explicit Congestion Notification (ECN) processing to estimate the fraction of bytes that encounter congestion, rather than simply detecting that some congestion has occurred. DCTCP then scales the TCP congestion window based on this estimate. This method achieves high burst tolerance, low latency, and high throughput with shallow-buffered switches.

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 8, 2016.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction 2
2. Terminology 4
3. DCTCP Algorithm 4
  3.1. Marking Congestion on the Switch 4
  3.2. Echoing Congestion Information on the Receiver 4
  3.3. Processing Congestion Indications on the Sender 5
  3.4. Handling of SYN, SYN-ACK, RST Packets 7
4. Implementation Issues 7
5. Deployment Issues 8
6. Known Issues 9
7. Implementation Status 10
8. Security Considerations 10
9. IANA Considerations 10
10. Acknowledgements 10
11. References 11
  11.1. Normative References 11
  11.2. Informative References 11
Authors' Addresses 12

1. Introduction

Large datacenters necessarily need a large number of network switches to interconnect the servers in the datacenter. Therefore, a datacenter can greatly reduce its capital expenditure by leveraging low cost switches. However, low cost switches tend to have limited queue capacities and thus are more susceptible to packet loss due to congestion.

Network traffic in the datacenter is often a mix of short and long flows, where the short flows require low latency and the long flows require high throughput. Datacenters also experience incast bursts,

where many endpoints send traffic to a single server at the same time. For example, this is a natural consequence of MapReduce algorithms. The worker nodes complete at approximately the same time, and all reply to the master node concurrently.

These factors place some conflicting demands on the queue occupancy of a switch:

- o The queue must be short enough that it does not impose excessive latency on short flows.
- o The queue must be long enough to buffer sufficient data for the long flows to saturate the path bandwidth.
- o The queue must be short enough to absorb incast bursts without excessive packet loss.

Standard TCP congestion control [RFC5681] relies on segment loss to detect congestion. This does not meet the demands described above. First, the short flows will start to experience unacceptable latencies before packet loss occurs. Second, by the time TCP congestion control kicks in on the sender, most of the incast burst has already been dropped.

[RFC3168] describes a mechanism for using Explicit Congestion Notification (ECN) from the switch for early detection of congestion, rather than waiting for segment loss to occur. However, this method only detects the presence of congestion, not the extent. In the presence of mild congestion, the TCP congestion window is reduced too aggressively and unnecessarily affects the throughput of long flows.

Datacenter TCP (DCTCP) improvises upon traditional ECN processing by estimating the fraction of bytes that encounter congestion, rather than simply detecting that some congestion has occurred. DCTCP then scales the TCP congestion window based on this estimate. This method achieves high burst tolerance, low latency, and high throughput with shallow-buffered switches.

It is recommended that DCTCP be deployed in a datacenter environment where the endpoints and the switching fabric are under a single administrative domain. Deployment issues around coexistence of DCTCP and conventional TCP, and lack of a negotiating mechanism between sender and receiver, and possible mitigations are also discussed.

2. Terminology

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

3. DCTCP Algorithm

There are three components involved in the DCTCP algorithm:

- The switch (or other intermediate device on the network) detects congestion and sets the Congestion Encountered (CE) codepoint in the IP header.

- The receiver echoes the congestion information back to the sender using the ECN-Echo (ECE) flag in the TCP header.

- The sender reacts to the congestion indication by reducing the TCP congestion window (cwnd).

3.1. Marking Congestion on the Switch

The switch indicates congestion to the end nodes by setting the CE codepoint in the IP header as specified in Section 5 of [RFC3168]. For example, the switch may be configured with a congestion threshold. When a packet arrives at the switch and its queue length is greater than the congestion threshold, the switch sets the CE codepoint in the packet. For example, Section 3.4 of [DCTCP10] suggests threshold marking with a threshold K > (RTT * C)/7, where C is the sending rate in packets per second. However, the actual algorithm for marking congestion is an implementation detail of the switch and will generally not be known to the sender and receiver. Therefore, sender and receiver MUST NOT assume that a particular marking algorithm is implemented by the switching fabric.

3.2. Echoing Congestion Information on the Receiver

According to Section 6.1.3 of [RFC3168], the receiver sets the ECE flag if any of the packets being acknowledged had the CE code point set. The receiver then continues to set the ECE flag until it receives a packet with the Congestion Window Reduced (CWR) flag set. However, the DCTCP algorithm requires more detailed congestion information. In particular, the sender must be able to determine the number of sent bytes that encountered congestion. Thus, the scheme described in [RFC3168] does not suffice.

One possible solution is to ACK every packet and set the ECE flag in the ACK if and only if the CE code point was set in the packet being

acknowledged. However, this prevents the use of delayed ACKs, which are an important performance optimization in datacenters.

Instead, DCTCP introduces a new Boolean TCP state variable, DCTCP Congestion Encountered (DCTCP.CE), which is initialized to false and stored in the Transmission Control Block (TCB). When sending an ACK, the ECE flag MUST be set if and only if DCTCP.CE is true. When receiving packets, the CE codepoint MUST be processed as follows:

1. If the CE codepoint is set and DCTCP.CE is false, send an ACK for any previously unacknowledged packets and set DCTCP.CE to true.

2. If the CE codepoint is not set and DCTCP.CE is true, send an ACK for any previously unacknowledged packets and set DCTCP.CE to false.

3. Otherwise, ignore the CE codepoint.

3.3. Processing Congestion Indications on the Sender

The sender estimates the fraction of sent bytes that encountered congestion. The current estimate is stored in a new TCP state variable, DCTCP.Alpha, which is initialized to 1 and MUST be updated as follows:

DCTCP.Alpha = DCTCP.Alpha * (1 - g) + g * M

where

- g is the estimation gain, a real number between 0 and 1. The selection of g is left to the implementation. See Section 4 for further considerations.

- M is the fraction of sent bytes that encountered congestion during the previous observation window, where the observation window is chosen to be approximately the Round Trip Time (RTT). In particular, an observation window ends when all the sent bytes in flight at the beginning of the window have been acknowledged.

In order to update DCTCP.Alpha, the TCP state variables defined in [RFC0793] are used, and three additional TCP state variables are introduced:

- DCTCP.WindowEnd: The TCP sequence number threshold for beginning a new observation window; initialized to SND.UNA.

- DCTCP.BytesSent: The number of bytes sent during the current observation window; initialized to zero.

o DCTCP.BytesMarked: The number of bytes sent during the current observation window that encountered congestion; initialized to zero.

The congestion estimator on the sender MUST process acceptable ACKs as follows:

1. Compute the bytes acknowledged (TCP SACK options [RFC2018] are ignored):

   BytesAcked = SEG.ACK - SND.UNA

2. Update the bytes sent:

   DCTCP.BytesSent += BytesAcked

3. If the ECE flag is set, update the bytes marked:

   DCTCP.BytesMarked += BytesAcked

4. If the sequence number is less than or equal to DCTCP.WindowEnd, then stop processing. Otherwise, the end of the observation window was reached, so proceed to update the congestion estimate as follows:

5. Compute the congestion level for the current observation window:

   M = DCTCP.BytesMarked / DCTCP.BytesSent

6. Update the congestion estimate:

   DCTCP.Alpha = DCTCP.Alpha * (1 - g) + g * M

7. Determine the end of the next observation window:

   DCTCP.WindowEnd = SND.NXT

8. Reset the byte counters:

   DCTCP.BytesSent = DCTCP.BytesMarked = 0

Rather than always halving the congestion window as described in [RFC3168], when the sender receives an indication of congestion (ECE), the sender MUST update cwnd as follows:

cwnd = cwnd * (1 - DCTCP.Alpha / 2)

Thus, when no sent byte experienced congestion, DCTCP.Alpha equals zero, and cwnd is left unchanged. When all sent bytes experienced congestion, DCTCP.Alpha equals one, and cwnd is reduced by half. Lower levels of congestion will result in correspondingly smaller reductions to cwnd.

Just as specified in [RFC3168], TCP should not react to congestion indications more than once every window of data. The setting of the "Congestion Window Reduced" (CWR) bit is also exactly as per [RFC3168].

3.4. Handling of SYN, SYN-ACK, RST Packets

[RFC3168] requires that compliant TCP MUST NOT set ECT on SYN or SYN-ACK packets. [RFC5562] proposes setting ECT on SYN-ACK packets, but maintains the restriction of no ECT on SYN packets. Both these RFCs prohibit ECT in SYN packets due to security concerns regarding malicious SYN packets with ECT set. These RFCs, however, are intended for general Internet use, and do not directly apply to a controlled datacenter deployment. The switching fabric can drop TCP packets that do not have the ECT set in the IP header. If SYN and SYN-ACK packets for DCTCP connections are non-ECT they will be dropped with high probability. For DCTCP connections the sender SHOULD set ECT for SYN, SYN-ACK and RST packets.

4. Implementation Issues

As noted in Section 3.3, the implementation must choose a suitable estimation gain. [DCTCP10] provides a theoretical basis for selecting the gain. However, it may be more practical to use experimentation to select a suitable gain for a particular network and workload. The Microsoft implementation of DCTCP in Windows Server 2012 uses a fixed estimation gain of 1/16.

The implementation must also decide when to use DCTCP. Datacenter servers may need to communicate with endpoints outside the datacenter, where DCTCP is unsuitable or unsupported. Thus, a global configuration setting to enable DCTCP will generally not suffice. DCTCP may be configured based on the IP address of the remote endpoint. Microsoft Windows Server 2012 also supports automatic selection of DCTCP if the estimated RTT is less than 10 msec and ECN is successfully negotiated, under the assumption that if the RTT is low, then the two endpoints are likely on the same datacenter network.

To prevent incast throughput collapse the minimum RTO (MinRTO) used by TCP should be lowered significantly. The default value of MinRTO in Windows is 300 msec which is much greater than the maximum

latencies inside a datacenter. Server 2012 onwards the MinRTO value is configurable allowing values as low as 10 msec on a per subnet or per TCP port basis or even globally. A lower MinRTO value requires corresponding a lower delayed ACK timeout on the receiver. It is recommended that the implementation allow configuration of lower timeouts for DCTCP connections.

In the same vein, it is also recommended that the implementation allow configuration of restarting the cwnd of idle DCTCP connections as described in [RFC5681] since network conditions change rapidly in the datacenter. The implementation can also allow configuration for discarding the value of DCTCP.Alpha after cwnd restart and timeouts.

[RFC3168] forbids the ECN-marking of pure ACK packets because of the inability of TCP to mitigate ACK-path congestion and protocol-wise preferential treatment by routers. However dropping pure ACKs rather than ECN marking them is disadvantageous in traffic scenarios typical in the datacenter. Because of the prevalence of bursty traffic patterns which involve transient congestion, the dropping of ACKS causes subsequent retransmission. It is recommended that the implementation a configuration knob that forces ECT on TCP pure ACK packets.

5. Deployment Issues

DCTCP and conventional TCP congestion control does not coexist well. In DCTCP, the marking threshold is set very low value to reduce queueing delay, thus a relatively small amount of congestion will exceed the marking threshold. During such periods of congestion, conventional TCP will suffer packet losses and quickly scale back cwnd. DCTCP, on the other hand, will use the fraction of marked packets to scale back cwnd. Thus rate reduction in DCTCP will be much lower than that of conventional TCP, and DCTCP traffic will dominate conventional TCP traffic traversing the same link. Hence if the traffic in the datacenter is a mix of conventional TCP and DCTCP, it is recommended that DCTCP traffic be segregated from conventional TCP traffic. [MORGANSTANLEY] describes a deployment that uses IP DSCP bits where AQM is applied to DCTCP traffic, while TCP traffic is managed via drop-tail queueing.

Today’s commodity switches allow configuration of a different marking/drop profile for non-TCP and non-IP packets. Non-TCP and non-IP packets should be able to pass through the switch unless the switch is really out of buffers. If the traffic in the datacenter consists of such traffic (including UDP), one possible mitigation would be to mark IP packets as ECT even when there is no transport that is reacting to the marking.

Since DCTCP relies on congestion marking by the switch, DCTCP can only be deployed in datacenters where the network infrastructure supports ECN. The switches may also support configuration of the congestion threshold used for marking. The proposed parameterization can be configured with switches that implement RED. [DCTCP10] provides a theoretical basis for selecting the congestion threshold, but as with estimation gain, it may be more practical to rely on experimentation or simply to use the default configuration of the device. DCTCP will degrade to loss-based congestion control when transiting a congested drop-tail link.

DCTCP requires changes on both the sender and the receiver, so both endpoints must support DCTCP. Furthermore, DCTCP provides no mechanism for negotiating its use, so both endpoints must be configured through some out-of-band mechanism to use DCTCP. A variant of DCTCP that can be deployed unilaterally and only requires standard ECN behavior has been described in [ODCTCP][BSDCAN], but requires additional experimental evaluation.

6. Known Issues

DCTCP relies on the sender's ability to reconstruct the stream of CE codepoints received by the remote endpoint. To accomplish this, DCTCP avoids using a single ACK packet to acknowledge segments received both with and without the CE codepoint set. However, if one or more ACK packets are dropped, it is possible that a subsequent ACK will cumulatively acknowledge a mix of CE and non-CE segments. This will, of course, result in a less accurate congestion estimate. There are some potential mitigations:

- o Even with a degraded congestion estimate, DCTCP may still perform better than [RFC3168].
- o If the estimation gain is small relative to the packet loss rate, the estimate may not be degraded much.
- o If packet losses mostly occur under heavy congestion, most drops will occur during an unbroken string of CE packets, and the estimate will be unaffected.

However, the affect of packet drops on DCTCP under real world conditions has not been analyzed.

DCTCP provides no mechanism for negotiating its use. Thus, there is additional management and configuration overhead required to ensure that DCTCP is not used with non-DCTCP endpoints. The affect of using DCTCP with a standard ECN endpoint has been analyzed in [ODCTCP][BSDCAN]. Furthermore, it is possible that other

implementations may also modify [RFC3168] behavior without negotiation, causing further interoperability issues.

Much like standard TCP, DCTCP is biased against flows with longer RTTs. A method for improving the fairness of DCTCP has been proposed in [ADCTCP], but requires additional experimental evaluation.

7. Implementation Status

This section documents the implementation status of the specification in this document, as recommended by [RFC6982].

This document describes DCTCP as implemented in Microsoft Windows Server 2012. Since publication of the first versions of this document, the Linux [LINUX] and FreeBSD [FREEBSD] operating systems have also implemented support for DCTCP in a way that is believed to follow this document.

8. Security Considerations

DCTCP enhances ECN and thus inherits the security considerations discussed in [RFC3168]. The processing changes introduced by DCTCP do not exacerbate these considerations or introduce new ones. In particular, with either algorithm, the network infrastructure or the remote endpoint can falsely report congestion and thus cause the sender to reduce cwnd. However, this is no worse than what can be achieved by simply dropping packets.

9. IANA Considerations

This document has no actions for IANA.

10. Acknowledgements

The DCTCP algorithm was originally proposed and analyzed in [DCTCP10] by Mohammad Alizadeh, Albert Greenberg, Dave Maltz, Jitu Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan.

Lars Eggert has received funding from the European Union's Horizon 2020 research and innovation program 2014-2018 under grant agreement No. 644866 ("SSICLOPS"). This document reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

11. References

11.1. Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

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

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

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, September 2001.

11.2. Informative References

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

[RFC5562] Kuzmanovic, A., Mondal, A., Floyd, S., and K. Ramakrishnan, "Adding Explicit Congestion Notification (ECN) Capability to TCP's SYN/ACK Packets", RFC 5562, June 2009.

[RFC6982] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", RFC 6982, July 2013.

[DCTCP10] Alizadeh, M., Greenberg, A., Maltz, D., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and M. Sridharan, "Data Center TCP (DCTCP)", Proc. ACM SIGCOMM 2010 Conference (SIGCOMM 10), August 2010, <http://dl.acm.org/citation.cfm?doid=1851182.1851192>.

[ODCTCP] Kato, M., "Improving Transmission Performance with One-Sided Datacenter TCP", M.S. Thesis, Keio University, 2014, <http://eggert.org/students/kato-thesis.pdf>.

[BSDCAN] Kato, M., Eggert, L., Zimmermann, A., van Meter, R., and H. Tokuda, "Extensions to FreeBSD Datacenter TCP for Incremental Deployment Support", BSDCan 2015, June 2015, <https://www.bsdcan.org/2015/schedule/events/559.en.html>.

[ADCTCP] Alizadeh, M., Javanmard, A., and B. Prabhakar, "Analysis of DCTCP: Stability, Convergence, and Fairness", Proc. ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 11), June 2011, <https://dl.acm.org/citation.cfm?id=1993753>.

[LINUX] Borkmann, D. and F. Westphal, "Linux DCTCP patch", 2014, <https://git.kernel.org/cgit/linux/kernel/git/davem/net-next.git/commit/?id=e3118e8359bb7c59555aca60c725106e6d78c5ce>.

[FREEBSD] Kato, M. and H. Panchasara, "DCTCP (Data Center TCP) implementation", 2015, <https://github.com/freebsd/freebsd/commit/8ad879445281027858a7fa706d13e458095b595f>.

[MORGANSTANLEY] Judd, G., "Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter", Proc. 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), May 2015, <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/judd>.

Authors' Addresses

Stephen Bensley
Microsoft
One Microsoft Way
Redmond, WA 98052
USA

Phone: +1 425 703 5570
Email: sbens@microsoft.com

Lars Eggert
NetApp
Sonnenallee 1
Kirchheim 85551
Germany

Phone: +49 151 120 55791
Email: lars@netapp.com
URI: http://eggert.org/

Dave Thaler
Microsoft

Phone: +1 425 703 8835
Email: dthaler@microsoft.com

Praveen Balasubramanian
Microsoft

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

Glenn Judd
Morgan Stanley

Phone: +1 973 979 6481
Email: glenn.judd@morganstanley.com

TCPM Working Group O. Bonaventure
Internet-Draft O. Tilmans
Intended status: Experimental UCLouvain
Expires: January 21, 2016 July 20, 2015

Analysis of the TCP EDO option
draft-bonaventure-tcpm-edo-analysis-00

Abstract

This document analyses how the proposed TCP EDO option copes with various types of middlebox interference.

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 21, 2016.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction 2
2. Middlebox Interference 3
2.1. Negotiating EDO 4
2.2. Replacement of the EDO option with NOP 4
2.3. Removal of the EDO option 6
2.4. Data Injection 6
2.5. Segment Splitting 7
2.6. Segment Coalescing 8
2.7. Option Injection 10
2.8. Summary 11
3. Testing the deployability of EDO with tracebox 13
3.1. Crafting packets with the EDO options 13
3.1.1. EDOREQUEST 13
3.1.2. EDO 13
3.1.3. EDOEXT 14
3.2. Evaluating the deployability of EDO with tracebox tests . 14
3.2.1. Example of test outputs 14
4. Security considerations 18
5. Conclusion 18
6. Acknowledgements 19
7. References 19
7.1. Normative References 19
7.2. Informative References 19
Authors' Addresses 20

1. Introduction

TCP [RFC0793] has probably been more successful than what its designers initially expected. TCP was designed with extensibility in mind thanks to a variable length header. The Data Offset (DO) field in the TCP header indicates the boundary between the extended TCP header and the segment payload. The DO is specified in blocks of 32 bits, which forces TCP implementations to pad the TCP header on 32 bits boundaries. The maximum value of the DO field is 15; which limits the length of the extended TCP header to 60 bytes (including the 20 bytes standard TCP header).

Various solutions have been proposed to cope with this limited extensibility of the TCP header (see [I-D.ietf-tcpm-tcp-edo] and the references cited in this document). A prototype implementation of the EDO proposal has been recently released [EDOLinux].However, as of this writing it has not been tested on the global Internet.

In this document we built upon previous work [IMC11] [HotMiddlebox13] [IMC13] and experience with Multipath TCP [RFC6824] to provide a qualitative analysis of the interactions between a TCP implementation

supporting the EDO proposal [I-D.ietf-tcpm-tcp-edo] and different types of middleboxes.

This document is organised as follows. We qualitatively analyse in Section 2 the impact of different types of middlebox interference on the EDO proposal [I-D.ietf-tcpm-tcp-edo]. Then, in Section 3, we show how tracebox [tracebox] could be used to perform a detailed evaluation of the interactions between EDO and real middleboxes.

2. Middlebox Interference

In this section, we analyse how the TCP EDO proposals deals with several of the types of middlebox interference that were identified in [IMC11] and guided the design of Multipath TCP [RFC6824].

In our analysis, we use the following graphical representation for the TCP segments. The first box represents the standard 20 bytes TCP header with the value of the DO field. The second box represents the EDO option (simple variant) with the header length set to 7 32 bits words. These two boxes are part of the extended TCP header. The third box is the TCP option that is encoded inside the payload. The number between parentheses is the length of the options in bytes. Finally, the last box contains the user data and its length in bytes between parentheses.

```
<--    TCP Header    --> <---     Segment Payload ------>

+-----------+-----------+============+=================+
| H (DO=6)  | EDO (L=7) | TCP Opt (4) | User Data (6)   |
+-----------+-----------+============+=================+
```

Figure 1: Simple EDO option

With the second variant of the EDO option that includes the segment length verification, the figure becomes.

```
<--       TCP Header      --> <---     Segment Payload ------>

+-----------+----------------+============+=================+
| H (DO=7)  | EDO (L=8,S=36) | TCP Opt (4) | User Data (6)   |
+-----------+----------------+============+=================+
```

Figure 2: EDO option with Segment Length Verification

Note that in the above figure, two NOP options are missing. These options should appear immediately after the EDO option to pad the

extended TCP header to a 32 bits word boundary. We omit these NOP options used for padding to simplify the figures in this document.

2.1. Negotiating EDO

The utilisation of the EDO option needs to be negotiated by placing the EDO Supported option in the SYN and SYN+ACK segments. The negotiation proposed in [I-D.ietf-tcpm-tcp-edo] raises two concerns based on the experience with such a negotiation for Multipath TCP [RFC6824].

First, using exactly the same option in the SYN and SYN+ACK segment is risky because there are some middleboxes [IMC13] that simply echo in the SYN+ACK any unknown option that they receive in the SYN. This is an incorrect, but unfortunately deployed behaviour.

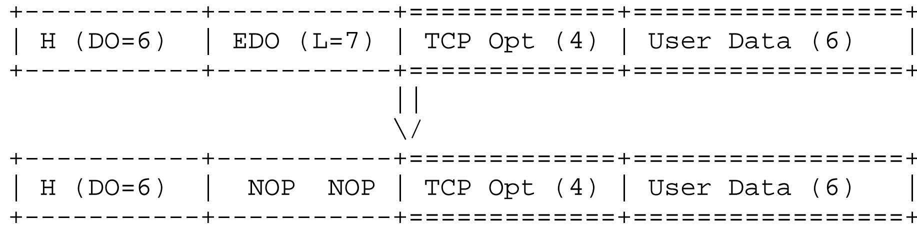
Second, [I-D.ietf-tcpm-tcp-edo] uses two different TCP option types : EDO Supported in the SYN and EDO Extension in the other segments. The initial design of Multipath TCP also used different option types but it then opted for a single option type and different subtypes. This change was motivated by the fact that if a middlebox accepts option type 'x' in the SYN, it will likely also accept it in the other segments since the handling of TCP options some middleboxes is often configured on a per-type basis [Normalizer]. However, accepting option type 'x' in the SYN is not a guarantee for the acceptance of option type 'y' in other segments.

In the remainder of this document, we only analyse the impact of middlebox interference on TCP segments that contain both options and data.

2.2. Replacement of the EDO option with NOP

The first middlebox interference that we consider is a middlebox that replaces the EDO option with NOPs [HotMiddlebox13] [Normalizer]. Replacing an unknown option with a NOP is a frequently used solution by middlebox vendors because it does not require any modification to the segment length. This is illustrated in Figure 3 for the simple EDO option.

Initial segment



Modified segment

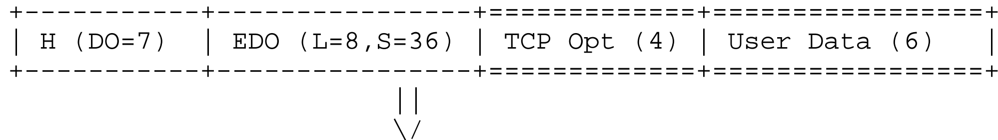
Figure 3: Simple EDO option through a middlebox that replaces EDO with NOP

Upon reception of the modified segment, the receiver will ignore the segment because it does not contain the EDO option. Unfortunately, if the sender retransmits the same segment, it is likely that the same modification will happen and the modified segment will again be lost. This could cause a blackhole where all segments sent by the sender are discarded by the receiver.

If the receiver accepts the segment without the EDO option, the situation is worse. Upon reception of the modified segment, the receiver will consider that the 4 bytes of the TCP options are part of the user data. If the segment was received in sequence, the receiver will acknowledge more data than the data sent by the sender.

If the EDO option includes the segment length verification, the same problem occurs.

Initial segment



Modified segment

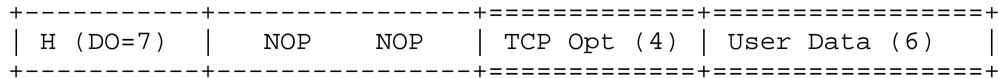


Figure 4: EDO option with Segment Length Verification through a middlebox that replaces EDO with NOP

2.3. Removal of the EDO option

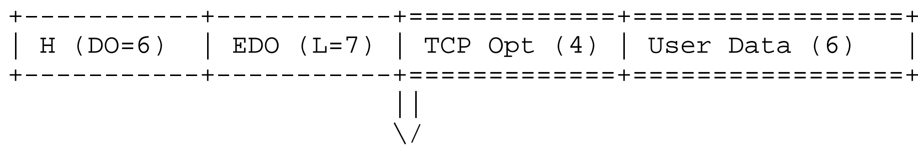
If a middlebox removes the EDO option [IMC11], then the receiver will act as explained in the previous section.

2.4. Data Injection

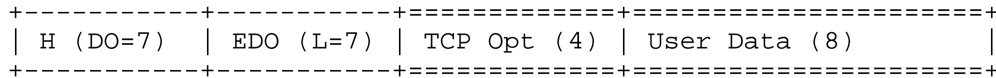
Middleboxes such as NAT boxes that include an ALG for protocols such as FTP are known to insert or remove data inside the payload of some segments.

Figure 5 shows the impact of such a middlebox on a TCP segment that contains the EDO option. The modified segment will be received and processed correctly (including the injected data) by the receiver.

Initial segment



```
+-----------+-----------+============+=================+
| H (DO=6)  | EDO (L=7) | TCP Opt (4) | User Data (6)   |
+-----------+-----------+============+=================+
                        ||
                        \/
```



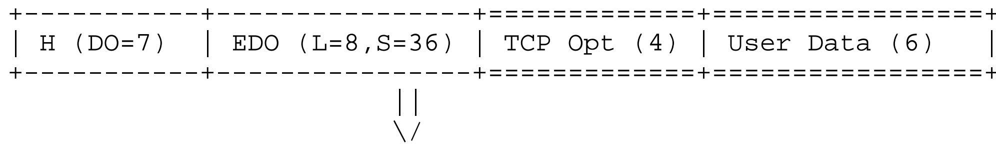
```
+-----------+-----------+============+=====================+
| H (DO=7)  | EDO (L=7) | TCP Opt (4) | User Data (8)       |
+-----------+-----------+============+=====================+
```

Modified segment

Figure 5: EDO option through a middlebox that injects data

If the EDO option includes the segment length, the situation is different. Figure 6 provides a graphical representation of the scenario. When the receiver parses the EDO option, it detects that the segment does not have the correct length and silently discards it. In this case, the same problem as described in Section 2.2 happens.

Initial segment



Modified segment

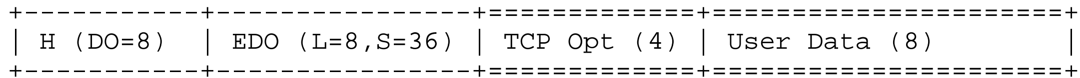
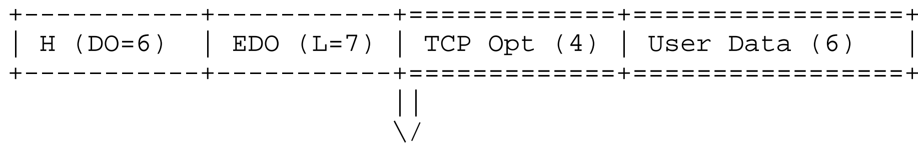
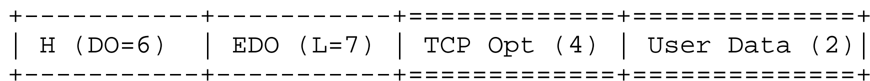


Figure 6: EDO option with Segment Length Verification through a middlebox that injects data

2.5. Segment Splitting

We now consider a middlebox that splits segments. Such a middlebox will usually copy the TCP options in the extended TCP header of the splitted segments. [IMC11] notes "All the NICs we tested correctly copied the options to all the split segments. TSO is now sufficiently commonplace so that designers of extensions to TCP should assume it." Segment splitting is illustrated in Figure 7.





and

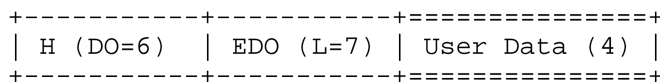


Figure 7: EDO option through a middlebox that splits segments

In this case, the receiver will correctly process the option in the payload of the first segment and pass the first two bytes of the user data to the application. However, in the second segment, the EDO option indicates that the first four bytes of the payload contain TCP

options. This part of the data will not be acknowledged by the receiver and will probably be retransmitted by the sender.

With the EDO option that includes the segment length information, the situation is different as shown in Figure 8. The receiver detects immediately that the segment has an invalid length and silently discards them and the same problem as described in Section 2.2 happens.

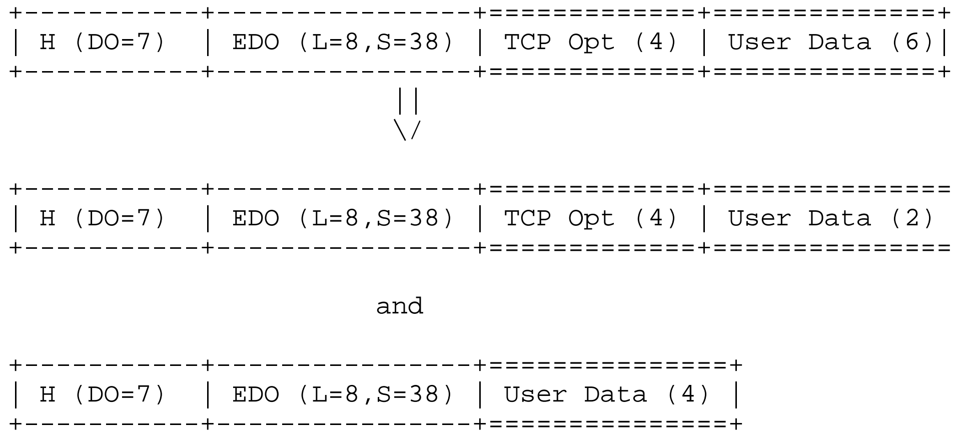


Figure 8: EDO option with Segment Length Verification through a middlebox that splits segments

2.6. Segment Coalescing

Our last middleboxes coalesces successive segments. We assume that the middlebox only coalesces the segments that contain the same option in the (extended) TCP header. This is inline with [IMC11] which notes after having tested segments with both a known and an unknown option : "For both option kinds, packets were coalesced only when their option values are same. The coalesced segment has one of the options on the original segments."

In our example, shown in Figure 9, the two segments contain the same EDO option but different TCP options in the payload. Due to the coalescing, the receiver will pass to the user application the second TCP option inside the payload.

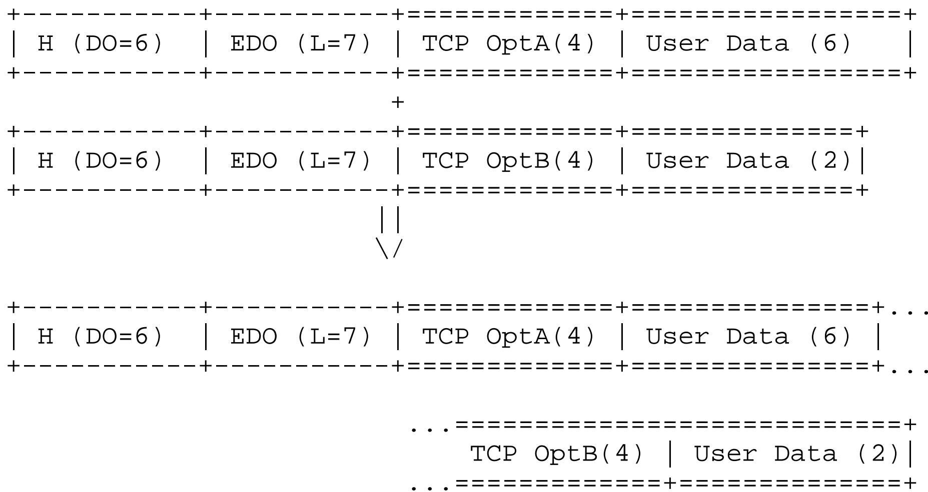


Figure 9: EDO option through a middlebox that coalesces segments

With the EDO option that includes the segment length, the receiver will silently discard the coalesced segment. However, as explained earlier, this may result in a deadlock where the sender retransmits segments that are never acknowledged by the receiver.

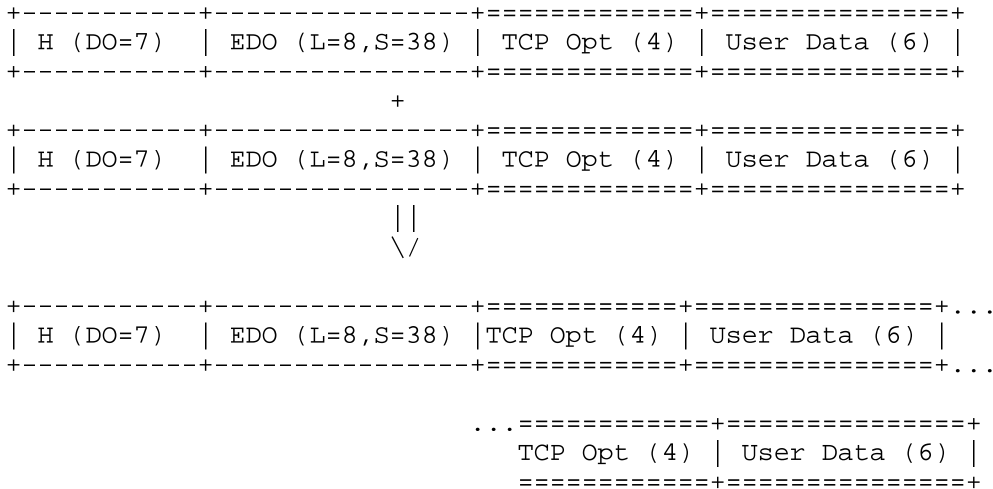


Figure 10: EDO option with Segment Length Verification through a middlebox that coalesces segments

2.7. Option Injection

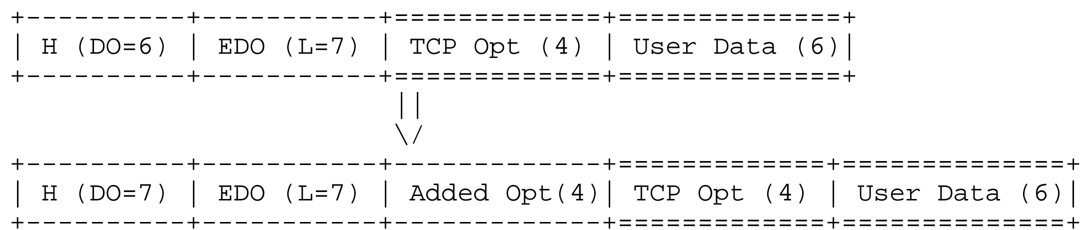
Finally, let us now consider what happens when a middlebox inserts an option inside the TCP header. For simplicity, we add an option that has a length of 4 bytes. We have no evidence of such behaviour, but it is possible in environments where middleboxes that operate in pair as shown in Figure 11. Several middelbox vendors have defined options that are used in the SYN segment to discover the presence of a downstream middlebox [I-D.ananth-middisc-tcpopt] and some vendors have defined specific TCP options that are used between such middleboxes. If the upstram upstream middlebox inserts an option, this option could be removed or replaced by NOP on the downstream middlebox.

```
client ---- mbox1 ---------- mbox2 ----- server
```

Figure 11: Cooperating middelboxes

The scenario with this middlebox is illustrated in Figure 12.

Initial segment



Modified segment

Figure 12: Simple EDO option through a middlebox that inserts an option

In this case, the receiver receives a strange segment. The DO field of the TCP header indicates that the extended header contains :

- o the EDO option
- o the Added option

The EDO option indicates that the extended header has a length of 28 bytes. This implies that it contains the following options :

o the EDO option

o the Added option

The four bytes TCP option that is included in the TCP payload is not detected as an option by the receiver based on the EDO option. This option is thus included in the data passed to the application.

With the TCP EDO option that includes the segment length information, the receiver can detect that the segment has been modified and silently discards it. Unfortunately, it is likely that when the sender retransmits the segment the same option is added to the retransmission. In this case, the same problem as described in Section 2.2 happens.

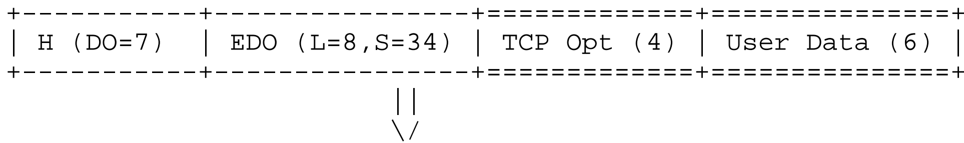


Figure 13: EDO option with Segment Length Verification through a middlebox that inserts an Added option

2.8. Summary

We summarise the main results of our qualitative analysis in two tables. First, Table 1 shows how the simple EDO option reacts with the different types of middlebox interference that have been discussed in this section. Table 2 shows the same information with the EDO option that includes the segment length information.

Middlebox Interference	Outcome
Replacement of EDO with NOP	silently discarded by receiver but risk of blackhole
Removal of EDO	silently discarded by receiver but risk of blackhole
Data injection	ok
Segment splitting	some data parsed as option and then likely retransmitted
Segment coalescing	option passed as user data in the bytestream
Option injection	option passed as user data in the bytestream

Table 1: Summary of Middelbox interference with the EDO extension

Middlebox Interference	Outcome
Replacement of EDO with NOP	silently discarded by receiver but risk of blackhole
Removal of EDO	silently discarded by receiver but risk of blackhole
Data injection	silently discarded by receiver but risk of blackhole
Segment splitting	silently discarded by receiver but risk of blackhole
Segment coalescing	silently discarded by receiver but risk of blackhole
Option injection	silently discarded by receiver but risk of blackhole

Table 2: Summary of Middelbox interference with the EDO extension with segment length validation

3. Testing the deployability of EDO with tracebox

The qualitative analysis described in the previous section provides some insights on several issues with the proposed EDO option. However, experience with Multipath TCP [IMC11] [HotMiddlebox13] [IMC13] shows that real measurements are required to understand the practical limits to the deployability of a TCP extension such as EDO.

In this section, we present how tracebox [tracebox] can help verifying the deployability of the proposed EDO extension. tracebox is a network diagnostic software that is similar to traceroute except that it allows to send any type of packet and support various options. tracebox detects middlebox interference by sending packets with increasing TTL/HopLimit values and analyses the ICMP replies returned by routers to infer modifications that were made on packets between two hops. tracebox works better through IPv6 routers that return the entire packet in the ICMP reply or IPv4 routers that implement the recommendation of [RFC1812] and quote the received packet in the return ICMP message. Furthermore, it can be extended through LUA scripts to support more advanced tests [traceboxlua].

3.1. Crafting packets with the EDO options

tracebox provides high-level functions to build packets that can then be used as probes for its default behavior, or sent/received to build more complex tests such as the ones involving data transfer after the TCP handshake. The latest version of tracebox provides bindings for all the three variants of the EDO option defined in [I-D.ietf-tcpm-tcp-edo].

3.1.1. EDOREQUEST

Implements the 2-bytes version of the option that is used in the three-way handshake.

Usage example: "IP / TCP / EDOREQUEST / sackp() / mpcapable()"

This builds a TCP SYN segment, advertising support for SACK, EDO and MPTCP.

3.1.2. EDO

Implements the 4-bytes extension containing the extended header length. When used, tracebox automatically adjusts the data offset field in the TCP header such that EDO is the last part of the header covered by the field.

Usage example: "IP / tcp{flags=TCP.ACK} / EDO / sack{{123, 456}, {789, 1011}} / NOP / NOP"

This builds a TCP header whose DO will be set to 6 (to cover only EDO), while the HeaderLength in the EDO extension will be set to 11 (44 bytes) (to include the SACK blocks in the payload)

3.1.3. EDOEXT

Implements the 6-bytes variant, which adds the segment length field.

Usage example: "IP / tcp{flags=TCP.ACK} / EDOEXT / NOP /NOP / sack{{123, 456}, {789, 1011}} / NOP / NOP"

This builds a TCP header whose DO will be set to 7, while the HeaderLength in the EDO extension will be set to 11 (46 bytes) and the SegmentLength field will be set to 46.

3.2. Evaluating the deployability of EDO with tracebox tests

The first set of tests is similar as to what has been done in [IMC11]. It uses tracebox to send packets containing one of the EDO variants with an increasing TTL and checks the ICMP payloads and the final reply from the target host to detect the eventual modifications that took place.

The second set of tests uses the Lua scripting capabilities of tracebox as well as its packet capture engine. In these tests, tracebox is used both on the client and on the server side. This enables to write a simple implementation of the TCP handshake, negociating EDO, then doing an actual data transfer where TCP options are included in the payload.

3.2.1. Example of test outputs

The test outputs shown in Figure 14 and Figure 15 reveal the different modifications that occur on a TCP SYN when being forwarded to either google.com or baidu.cn, and eventually the reply that we received from the server. Hops marked with the "[PARTIAL]" tags denotes routers not following the recommendations of [RFC1812] regarding ICMP replies. We can see that Google does not advertize back the support for the EDO extension (as expected ...) and that instead it added a TCP MSS option in the SYN+ACK (hop 20).

```
# tracebox -n -p 'IP/TCP/EDOREQUEST' www.google.com
tracebox to 74.125.70.105 (www.google.com): 64 hops max
1: 192.168.0.1  [PARTIAL]
2: *
3: 78.129.127.145  [PARTIAL] IP::TTL IP::CheckSum
4: 212.68.211.13  [PARTIAL] IP::TTL IP::CheckSum
5: 149.6.135.141 TCP::CheckSum IP::TTL IP::CheckSum
6: 154.54.39.50 TCP::CheckSum IP::DiffServicesCP IP::TTL
                IP::CheckSum
...
19: *
20: 74.125.70.105 TCP::SrcPort TCP::DstPort TCP::SeqNumber
    TCP::AckNumber TCP::Flags TCP::WindowsSize TCP::CheckSum
    IP::Identification IP::Flags IP::TTL IP::CheckSum IP::SourceIP
    IP::DestinationIP +TCPOptionMaxSegSize  -TCPOptionPad
    -TCPEDORequest
```

Figure 14: Sending TCP SYN advertizing EDO support.

The trace towards Baidu however exhibits the same behavior as what was reported in [IMC13]: EDO is an unkown option, but it is echo'ed back to the sender in the TCP SYN+ACK (unlike in the Google trace, there is no -TCPEDORequest modification that has been detected).

```
# tracebox -n -p 'IP/TCP/EDOREQUEST' www.baidu.com
tracebox to 103.235.46.39 (www.baidu.com): 64 hops max
1: 192.168.0.1  [PARTIAL]
2: *
3: 78.129.127.145  [PARTIAL] IP::TTL IP::CheckSum
4: 212.68.211.13  [PARTIAL] IP::TTL IP::CheckSum
5: 195.219.227.17  [PARTIAL] IP::TTL IP::CheckSum
6: 195.219.241.9 TCP::CheckSum IP::TTL IP::CheckSum

...

16: *
17: *
18: 103.235.46.39 TCP::SrcPort TCP::DstPort TCP::SeqNumber
    TCP::AckNumber  TCP::Flags TCP::WindowsSize TCP::CheckSum
    IP::TTL IP::CheckSum IP::SourceIP IP::DestinationIP
```

Figure 15: Sending TCP SYN advertizing EDO support.

Our second sample test is a stateful one. It consists in a client establishing a TCP connection towards a FTP server running on the standard port (21), and then sending a PORT command. Both the client and the server are implemented using tracebox Lua API, and visible in

Figure 16 and Figure 17. The client connection starts from a home network, behind a NAT, while the server runs in a datacenter. The client output is shown in Figure 18. The server-side output of the test is visible in Figure 19, and shows that an unconsistency due to the client NAT has been detected between the EDOEXT reported segment length and the actual segment length, which would cause the packet to be silently ignored by the server resulting in a blackhole.

```
iph = ip{dst=dn4('test1.multipath-tcp.org')}
syn = iph / tcp{dst=21} / EDOREQUEST
synack = syn:sendrecv{}

if not synack then
    print("Server did not reply...")
    return
end

if synack:tcp():getflags() ~= TCP.SYN + TCP.ACK then
    print("Server did not reply with SYN+ACK")
    return
end

if not synack:get(EDOREQUEST) then
    print('Server does not support EDO')
    return
end

-- build PORT probe
ip_port = syn:source():gsub("%.", ",")
data = iph / tcp{src=syn:tcp():getsource(),
                dst=21, seq=syn:tcp():getseq()+1,
                ack=synack:tcp():getseq()+1,
                flags=TCP.ACK}
          / EDOEXT
          / raw('PORT '.. ip_port .. ',189,68\r\n')
print('Sending: ' .. tostring(data:payload():data()))
data:send()
```

Figure 16: Lua code for a client establishing a connection to a FTP server supporting EDO

```
function receive(pkt)
    if pkt:tcp():getflags() == TCP.SYN then
        print('Client connected from ' .. pkt:source())
        if pkt:get(EDOREQUEST) then
            print('Client advertized EDO option')
            local synack = ip{dst=pkt:source()}
                           / tcp{src=21,
                                 dst=pkt:tcp():getsource(),
                                 ack=pkt:tcp():getseq() + 1,
                                 flags=TCP.SYN + TCP.ACK}
                           / EDOREQUEST / NOP / NOP
            synack:send()
        else
            print('No EDO option, ignoring client')
        end
    else
        local edoh = pkt:get(EDO)
        if edoh ~= nil edoh:length() == 6 then
            print('Received data: '
                  .. tostring(pkt:payload():data()))
            print('Packet has an extended EDO option')
            local seg_len = pkt:iplayer():payloadlen()
                            - edoh:headerlength() * 4
            print('Segment length is: ' .. seg_len)
            print('EDO Segment length is: '
                   .. edoh:segmentlength())
        end
    end
end

snif({'-p', 'tcp', '--dport', 21}, receive)
```

Figure 17: Lua code for a server listening on port 21 supporting EDO

```
# tracebox -s client_ftp_edo.lua test1.multipath-tcp.org
Sending: PORT 192,168,0,4,189,68
```

Figure 18: Client output for the EDO/FTP test

```
# tracebox -s server_ftp_eco.lua
Client connected from 85.27.48.241
Client advertized EDO option
Received data: PORT 85,27,48,241,189,68

Packet has an extended EDO option
Segment length is: 26
EDO Segment length is: 25
```

Figure 19: Server output for the EDO/FTP test

4. Security considerations

This document discusses the interference between the proposed EDO TCP option [I-D.ietf-tcpm-tcp-edo] and middleboxes. It does not affect the security of TCP.

5. Conclusion

In this document, we have analysed how the proposed EDO option can cope with various types of middlebox interferences. For the simple EDO extension, our analysis reveals that this option is not safe for several important types of middlebox interference. For the EDO extension that includes a segment length verification, the main risk is that when a receiver detects a segment that has been modified by a middlebox, that segment is silently discarded and there is no guarantee that a future retransmission of this segment will not again be discarded by the receiver. There is thus a risk of blackhole that does not appear acceptable for a TCP option that is intended to be used to support other TCP extensions.

The main problem with the current EDO proposal [I-D.ietf-tcpm-tcp-edo] is that it does not include any feedback mechanism to enable the receiver to inform the sender about either the correct operation of EDO or the detection of any invalid segment. Without such a feedback mechanism, it is unlikely that a variant of EDO would be able to cope with the different types of middlebox interference.

Finally, we have shown that tracebox can be used to perform tests with the proposed EDO option and we would encourage the TCPM working group to conduct such tests on a large scale before any final decision on EDO.

6. Acknowledgements

This work was partially supported by the FP7-Trilogy2 project.

7. References

7.1. Normative References

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

7.2. Informative References

[EDOLinux]
Trieu, H., Touch, J., and T. Faber, "Implementation of the TCP Extended Data Offset Option", ISI-TR-696 , n.d., <http://www.isi.edu/touch/pubs/isi-tr-2015-696.pdf>.

[HotMiddlebox13]
Hesmans, B., Duchene, F., Paasch, C., Detal, G., and O. Bonaventure, "Are TCP Extensions Middlebox-proof?", CoNEXT workshop HotMiddlebox , December 2013, <http://inl.info.ucl.ac.be/publications/are-tcp-extensions-middlebox-proof>.

[I-D.ananth-middisc-tcpopt]
Knutsen, A., Ramaiah, A., and A. Ramasamy, "TCP option for transparent Middlebox negotiation", draft-ananth-middisc-tcpopt-02 (work in progress), February 2013.

[I-D.ietf-tcpm-tcp-edo]
Touch, J. and W. Eddy, "TCP Extended Data Offset Option", draft-ietf-tcpm-tcp-edo-03 (work in progress), April 2015.

[IMC11] Honda, M., Nishida, Y., Raiciu, C., Greenhalgh, A., Handley, M., and H. Tokuda, "Is it still possible to extend TCP?", Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference (IMC ’11) , 2011, <http://doi.acm.org/10.1145/2068816.2068834>.

[IMC13] Detal, G., Hesmans, B., Bonaventure, O., Vanaubel, Y., and B. Donnet, "Revealing Middlebox Interference with tracebox", Proceedings of the 2013 ACM SIGCOMM conference on Internet measurement conference , 2013, <http://inl.info.ucl.ac.be/publications/revealing-middlebox-interference-tracebox>.

[Normalizer]
Cisco Systems, ., "Configuring TCP Normalization", 2015, <
http://www.cisco.com/c/en/us/td/docs/security/asa/asa82/co
nfiguration/guide/config/conns_tcpnorm.html#wp1084313>.

[RFC1812] Baker, F., Ed., "Requirements for IP Version 4 Routers",
RFC 1812, DOI 10.17487/RFC1812, June 1995,
<http://www.rfc-editor.org/info/rfc1812>.

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

[tracebox]
Detal, G. and O. Tilmans, "tracebox",
<http://www.tracebox.org>.

[traceboxlua]
Tilmans, O., "LUA bindings for tracebox", n.d.,
<http://www.tracebox.org/lua_doc/index.html>.

Authors' Addresses

Olivier Bonaventure
UCLouvain

Email: Olivier.Bonaventure@uclouvain.be

Olivier Tilmans
UCLouvain

Email: Olivier.Tilmans@uclouvain.be

tcpm M. HAWARI
Internet-Draft Ecole Polytechnique / Cisco Systems
Intended status: Experimental July 3, 2015
Expires: January 4, 2016

TCP Fast Open Cookie for IPv6 prefixes
draft-hawari-tcpm-tfo-ipv6-prefixes-00

Abstract

In order to support applications that require a large number of addresses or address space, a host may be assigned an aggregate IPv6 prefix rather than one or more discrete IPv6 addresses. This document briefly describes cases where this may occur, and specifies an extension to TCP Fast Open [RFC7413] to allow a client to use a single TCP Fast Open cookie for an IPv6 prefix.

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 4, 2016.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

Table of Contents

1. Introduction 2
1.1. Terminology 2
1.2. Motivation 2
2. Proposed extension to TCP Fast Open 3
2.1. Overview 3
2.2. The TCP Fast Open Prefix Length option (TFOPL). 3
2.3. Behaviour of the client 3
2.4. Behaviour of the server 3
3. TCP Fast Open Prefix Length Option Format 4
4. IANA Considerations 4
5. Security Considerations 4
6. Acknowledgements 4
7. Normative References 5
Author's Address 5

1. Introduction

1.1. Terminology

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

1.2. Motivation

TCP Fast Open(TFO) [RFC7413] provides applications with a mechanism allowing them to send data in the TCP SYN packet. It relies on a TFO cookie which is delivered by the server to the client. This cookie is then used by the client for all further TFO transactions to the server's IP address.

Unlike IPv4, IPv6 provides the applications with the ability to leverage a great number of addresses. A single server may be associated to a whole IP prefix from which each of the IP addresses may be mapped to various application layer concepts. For example an IPv6 address can identify a service, a process, or even a chunk of data.

Consequently, it is convenient to make a TFO cookie valid for a whole prefix instead of a single network address as this prefix may be hosted on a single server. That is the reason why this document proposes an extension to TFO to achieve this goal.

Another application of this extension appears in the context of a DNS-load-balanced cluster of servers. In this case, one server of the cluster will provide the client with a TFO cookie which is valid for the whole cluster, thus reducing the number of three way handshakes.

2. Proposed extension to TCP Fast Open

2.1. Overview

When a server receives a TCP SYN packet asking for a TFO cookie, it generates a TFO cookie and sends a TCP SYN+ACK with the TFO cookie and the prefix length for which the TFO cookie is valid, thus allowing the client to reuse the cookie for the whole prefix containing the server.

2.2. The TCP Fast Open Prefix Length option (TFOPL).

A new TCP Option named TCP Fast Open Prefix Length (TFOPL) is specified here. The value of this option is the prefix length (in bits) which a TCP Fast Open (TFO) cookie is valid for.

2.3. Behaviour of the client

When the client requests a TFO cookie, it MUST include in the TCP SYN packet a TFOPL Option containing the value zero and an empty TFO cookie.

When the client receives a TCP SYN+ACK packet including a TFO cookie and a TFOPL Option containing the value N, it MAY use this cookie for all further transactions with a host whose IP address belongs to the prefix of length N associated to the server.

When the client receives a TCP SYN+ACK packet including a TFO cookie without any TFOPL Option, it MUST assume that the received prefix length is 128 (i.e, a single address).

2.4. Behaviour of the server

When the server receives a TCP SYN packet including a TFO cookie and a TFOPL Option, it MUST generate a TFO cookie and reply with a SYN+ACK packet including the TFO cookie and the TFOPL option containing the prefix length in bits for which the TFO cookie is valid.

When the server receives a TCP SYN packet including a TFO cookie but without any TFOPL Option, it MUST have the same behaviour as a regular TFO-enabled server as described in [RFC7413].

If a TFO cookie is delivered by the server for a prefix of length N, all the servers belonging to this prefix MUST consider it as a valid TFO cookie, until its potential invalidation for the whole prefix. They MUST also deliver TFO cookies with a TFOPL option containing the value N.

3. TCP Fast Open Prefix Length Option Format

The TFOPL option has a TLV structure:

```
+------------+------------+-----------+
| TCP Option |   Length   |   Prefix  |
| Kind Number|            |   Length  |
| (1 BYTE)   |  (1 BYTE)  |  (1 BYTE) |
+------------+------------+-----------+
```

The length field MUST be equal to 0x3. The Prefix Length field MUST contain a value between 0 and 128. Any out of bounds Prefix Length Value MUST be considered as an absence of TFOPL option.

4. IANA Considerations

IANA is kindly asked to allocate a value in the "TCP Option Kind Numbers" registry for the TCP Fast Open Prefix Length Option. The length of this new TCP option is 3.

5. Security Considerations

A rogue TFO server can send a TFOPL option containing an arbitrary small value, thus endangering all the TFO cookies saved by the client prior to this transaction. The client may implement the following strategy in order to mitigate this issue:

The client has a parameter N between 0 and 128. This parameter is a lower bound for the received prefix lengths. When the client receives a SYN+ACK packet with a TFOPL containing a value N', it acts as if it has received min(N,N'). This strategy prevents the server from erasing the cookies of the client outside the own prefix of length N of the server. A typical value for N may be 64.

6. Acknowledgements

The author would like to thank Pierre Pfister, Mark Townsley and Andrew Yourtchencko for their contributions to this draft.

7. Normative References

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

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

Author's Address

Mohammed HAWARI
Ecole Polytechnique / Cisco Systems

Email: mohammed@hawari.fr

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: June 8, 2019
December 5, 2018

W. Eddy, Ed.
MTI Systems

Transmission Control Protocol Specification
draft-ietf-tcpm-rfc793bis-12

Abstract

This document specifies the Internet's Transmission Control Protocol (TCP). TCP is an important transport layer protocol in the Internet stack, and has continuously evolved over decades of use and growth of the Internet. Over this time, a number of changes have been made to TCP as it was specified in RFC 793, though these have only been documented in a piecemeal fashion. This document collects and brings those changes together with the protocol specification from RFC 793. This document obsoletes RFC 793, as well as 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 updates RFC 5961 due to a small clarification in reset handling while in the SYN-RECEIVED state.

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

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [4].

Status of This Memo

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

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

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

This Internet-Draft will expire on June 8, 2019.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1. Purpose and Scope 3
2. Introduction 4
  2.1. Key TCP Concepts 5
3. Functional Specification 5
  3.1. Header Format 6
  3.2. Terminology 11
  3.3. Sequence Numbers 15
  3.4. Establishing a connection 22
  3.5. Closing a Connection 29
    3.5.1. Half-Closed Connections 31
  3.6. Precedence and Security 32
  3.7. Segmentation 33

3.7.1. Maximum Segment Size Option 34
3.7.2. Path MTU Discovery 36
3.7.3. Interfaces with Variable MTU Values 36
3.7.4. Nagle Algorithm 37
3.7.5. IPv6 Jumbograms 37
3.8. Data Communication 37
3.8.1. Retransmission Timeout 38
3.8.2. TCP Congestion Control 38
3.8.3. TCP Connection Failures 39
3.8.4. TCP Keep-Alives 40
3.8.5. The Communication of Urgent Information 40
3.8.6. Managing the Window 41
3.9. Interfaces 46
3.9.1. User/TCP Interface 46
3.9.2. TCP/Lower-Level Interface 55
3.10. Event Processing 57
3.11. Glossary 82
4. Changes from RFC 793 87
5. IANA Considerations 92
6. Security and Privacy Considerations 92
7. Acknowledgements 93
8. References 94
8.1. Normative References 94
8.2. Informative References 95
Appendix A. Other Implementation Notes 99
A.1. IP Security Compartment and Precedence 99
A.2. Sequence Number Validation 99
A.3. Nagle Modification 100
A.4. Low Water Mark 100
Appendix B. TCP Requirement Summary 100
Author's Address 104

1. Purpose and Scope

In 1981, RFC 793 [12] was released, documenting the Transmission Control Protocol (TCP), and replacing earlier specifications for TCP that had been published in the past.

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

For several decades, RFC 793 plus a number of other documents have combined to serve as the specification for TCP [37]. Over time, a number of errata have been identified on RFC 793, as well as deficiencies in security, performance, and other aspects. A number of enhancements has grown and been documented separately. These were never accumulated together into an update to the base specification.

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

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

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

2. Introduction

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

This document describes the basic functionality expected in modern implementations of TCP, and replaces the protocol specification in RFC 793. It does not replicate or attempt to update the introduction and philosophy content in RFC 793 (sections 1 and 2 of that document). 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" [37] 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. [25]) 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 common TCP implementations today include the high-performance extensions in [35], but these are not strictly required or discussed in this document.

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

2.1. 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 of losses and errors 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 rerouting.

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 flow data only unidirectionally, if they so choose.

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

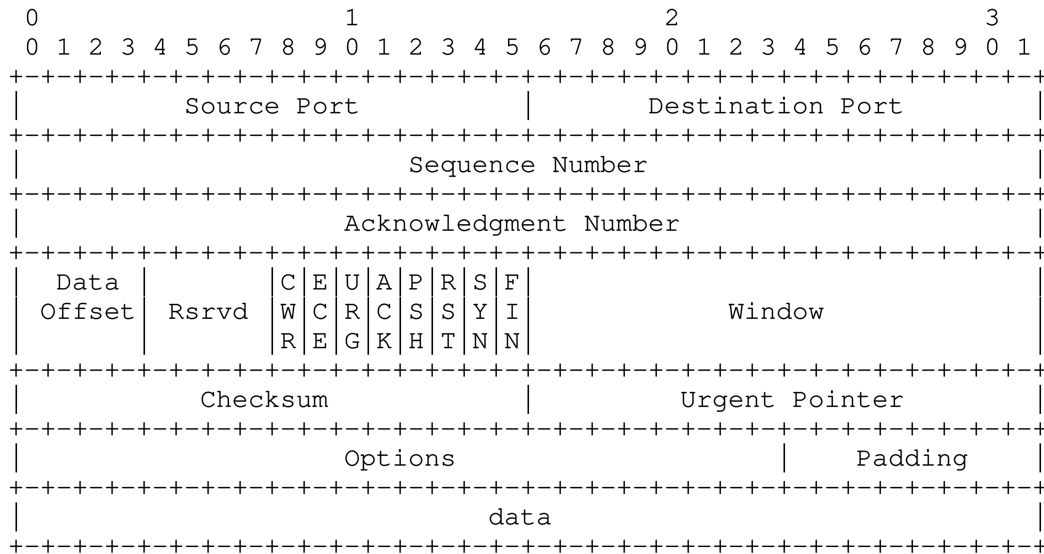
A more detailed description of TCP's features compared to other transport protocols can be found in Section 3.1 of [40]. Further description of the motivations for developing TCP and its role in the Internet stack can be found in Section 2 of [12] and earlier versions of the TCP specification.

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] [5]. A TCP header follows the Internet header, supplying information specific to the TCP protocol. This division allows for the existence of host level protocols other than TCP. In early development of the Internet suite of protocols, the IP header fields had been a part of TCP.

TCP Header Format



TCP Header Format

Note that one tick mark represents one bit position.

Figure 1

Source Port: 16 bits

The source port number.

Destination Port: 16 bits

The destination port number.

Sequence Number: 32 bits

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

Acknowledgment Number: 32 bits

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

Data Offset: 4 bits

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

Rsrvd - Reserved: 4 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.

Control Bits: 8 bits (from left to right):

CWR: Congestion Window Reduced (see [9])
ECE: ECN-Echo (see [9])
URG: Urgent Pointer field significant
ACK: Acknowledgment field significant
PSH: Push Function (see Paragraph 5)
RST: Reset the connection
SYN: Synchronize sequence numbers
FIN: No more data from sender

Window: 16 bits

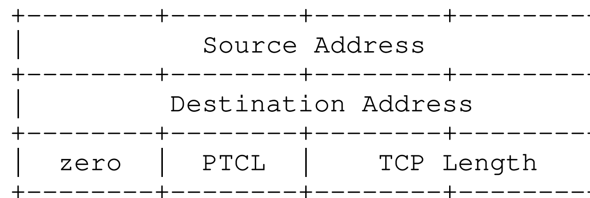
The number of data octets beginning with the one indicated in the acknowledgment field which the sender of this segment is willing to accept.

The window size MUST be treated as an unsigned number, or else large window sizes will appear like negative windows and TCP will now work (MUST-1). It is RECOMMENDED that implementations will reserve 32-bit fields for the send and receive window sizes in the connection record and do all window computations with 32 bits (REC-1).

Checksum: 16 bits

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

The checksum also covers a pseudo header conceptually prefixed to the TCP header. The pseudo header is 96 bits for IPv4 and 320 bits for IPv6. For IPv4, this pseudo header contains the Source Address, the Destination Address, the Protocol, and TCP length. This gives the TCP protection against misrouted segments. This information is carried in IPv4 and is transferred across the TCP/Network interface in the arguments or results of calls by the TCP on the IP.



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

For IPv6, the pseudo header is contained in section 8.1 of RFC 2460 [5], 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, 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 [41], and each option is defined in other RFCs, as indicated there. That set includes experimental options that can be extended to support multiple concurrent uses [34].

A given TCP implementation can support any currently defined options, but the following options MUST be supported (MUST-4) (kind indicated in octal):

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

A TCP MUST be able to receive a TCP option in any segment (MUST-5). A TCP MUST (MUST-6) ignore without error any TCP option it does not implement, assuming that the option has a length field (all TCP options except End of option list and No-Operation have length fields). TCP MUST be prepared to handle an illegal option length (e.g., zero) without crashing; a suggested procedure is to reset the connection and log the reason (MUST-7).

Specific Option Definitions

End of Option List

```
+--------+
|00000000|
+--------+
 Kind=0
```

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

No-Operation

```
+--------+
|00000001|
+--------+
 Kind=1
```

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

Maximum Segment Size (MSS)

```
+--------+--------+---------+--------+
|00000010|00000100|   max seg size   |
+--------+--------+---------+--------+
 Kind=2   Length=4
```

Maximum Segment Size Option Data: 16 bits

If this option is present, then it communicates the maximum receive segment size at the TCP which 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. If this option is not used, any segment size is allowed. A more complete description of this option is in Section 3.7.1.

Padding: variable

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

3.2. Terminology

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

Send Sequence Variables

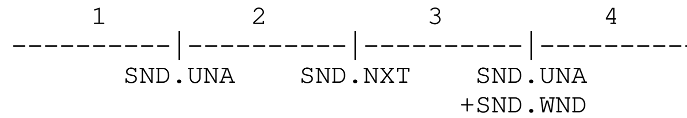
```
SND.UNA - send unacknowledged
SND.NXT - send next
SND.WND - send window
SND.UP  - send urgent pointer
SND.WL1 - segment sequence number used for last window update
SND.WL2 - segment acknowledgment number used for last window
          update
ISS     - initial send sequence number
```

Receive Sequence Variables

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

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

Send Sequence Space



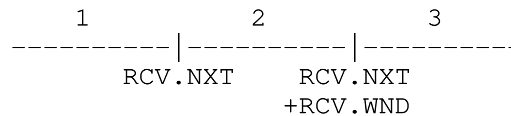
1 - old sequence numbers which have been acknowledged
2 - sequence numbers of unacknowledged data
3 - sequence numbers allowed for new data transmission
4 - future sequence numbers which are not yet allowed

Send Sequence Space

Figure 2

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

Receive Sequence Space



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

Receive Sequence Space

Figure 3

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

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

Current Segment Variables

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

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

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

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

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

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

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

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

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

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

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

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

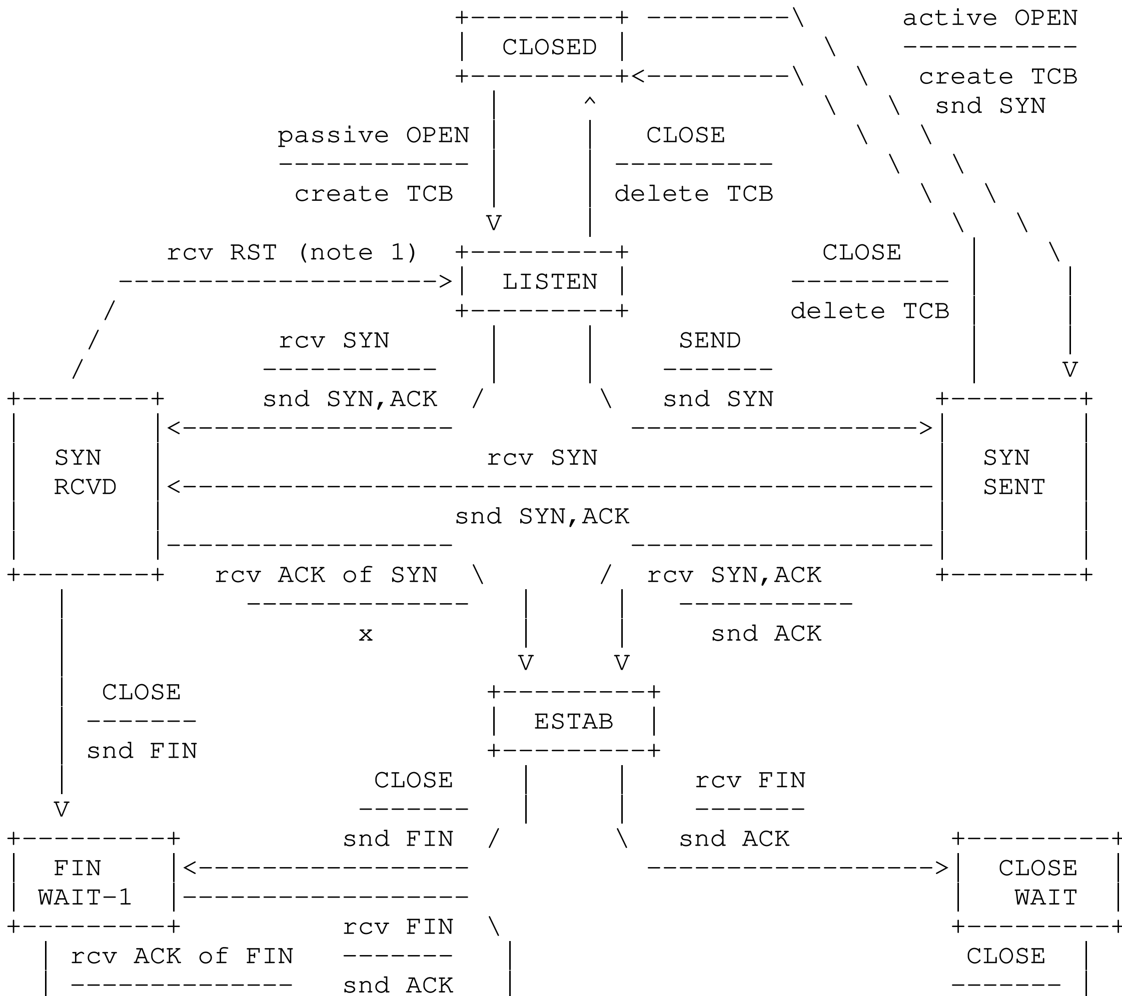
CLOSED - represents no connection state at all.

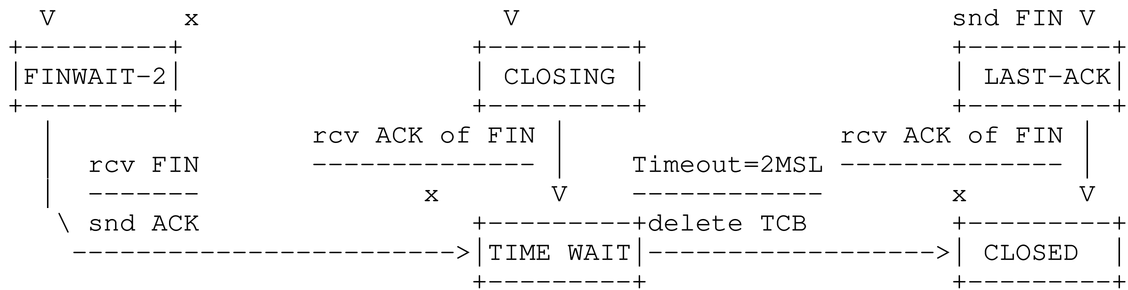
A TCP connection progresses from one state to another in response to events. The events are the user calls, OPEN, SEND, RECEIVE, CLOSE,

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

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





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.

TCP Connection State Diagram

Figure 4

3.3. Sequence Numbers

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

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

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

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

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

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

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

SND.UNA = oldest unacknowledged sequence number

SND.NXT = next sequence number to be sent

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

SEG.SEQ = first sequence number of a segment

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

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

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

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

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

When data is received the following comparisons are needed:

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

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

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

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

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

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

or

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

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

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

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

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

We have taken advantage of the numbering scheme to protect certain control information as well. This is achieved by implicitly including some control flags in the sequence space so they can be retransmitted and acknowledged without confusion (i.e., one and only one copy of the control will be acted upon). Control information is not physically carried in the segment data space. Consequently, we must adopt rules for implicitly assigning sequence numbers to control. The SYN and FIN are the only controls requiring this

protection, and these controls are used only at connection opening and closing. For sequence number purposes, the SYN is considered to occur before the first actual data octet of the segment in which it occurs, while the FIN is considered to occur after the last actual data octet in a segment in which it occurs. The segment length (SEG.LEN) includes both data and sequence space occupying controls. When a SYN is present then SEG.SEQ is the sequence number of the SYN.

Initial Sequence Number Selection

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

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

The recommended ISN generator is based on the combination of a (possibly fictitious) 32 bit clock whose low order bit is incremented roughly every 4 microseconds, and a pseudorandom hash function (PRF). The clock component is intended to insure that with a Maximum Segment Lifetime (MSL), generated ISNs will be unique, since it cycles approximately every 4.55 hours, which is much longer than the MSL. This recommended algorithm is further described in RFC 1948 and builds on the basic clock-driven algorithm from RFC 793.

A TCP MUST use a clock-driven selection of initial sequence numbers (MUST-8), and SHOULD generate its Initial Sequence Numbers with the expression:

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

where M is the 4 microsecond timer, and F() is a pseudorandom function (PRF) of the connection's identifying parameters ("localip, localport, remoteip, remoteport") and a secret key ("secretkey") (SHLD-1). F() MUST NOT be computable from the outside (MUST-9), or an attacker could still guess at sequence numbers from the ISN used

for some other connection. The PRF could be implemented as a cryptographic has of the concatenation of the TCP connection parameters and some secret data. For discussion of the selection of a specific hash algorithm and management of the secret key data, please see Section 3 of [32].

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

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

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

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

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

A three way handshake is necessary because sequence numbers are not tied to a global clock in the network, and TCPs may have different mechanisms for picking the ISN's. The receiver of the first SYN has no way of knowing whether the segment was an old delayed one or not, unless it remembers the last sequence number used on the connection (which is not always possible), and so it must ask the sender to verify this SYN. The three way handshake and the advantages of a clock-driven scheme are discussed in [46].

Knowing When to Keep Quiet

To be sure that a TCP does not create a segment that carries a sequence number which may be duplicated by an old segment remaining in the network, the TCP must keep quiet for an MSL before assigning any sequence numbers upon starting up or recovering from a crash in

which memory of sequence numbers in use was lost. For this specification the MSL is taken to be 2 minutes. This is an engineering choice, and may be changed if experience indicates it is desirable to do so. Note that if a TCP is reinitialized in some sense, yet retains its memory of sequence numbers in use, then it need not wait at all; it must only be sure to use sequence numbers larger than those recently used.

The TCP Quiet Time Concept

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

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

Under normal conditions, TCPs keep track of the next sequence number to emit and the oldest awaiting acknowledgment so as to avoid mistakenly using a sequence number over before its first use has been acknowledged. This alone does not guarantee that old duplicate data is drained from the net, so the sequence space has been made very large to reduce the probability that a wandering duplicate will cause trouble upon arrival. At 2 megabits/sec. it takes 4.5 hours to use up 2**32 octets of sequence space. Since the maximum segment lifetime in the net is not likely to exceed a few tens of seconds, this is deemed ample protection for foreseeable nets, even if data rates escalate to 10’s of megabits/sec. At 100 megabits/sec, the cycle time is 5.4 minutes which may be a little short, but still within reason.

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

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

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

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

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

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

3.4. Establishing a connection

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

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

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

```
    TCP A                                                TCP B

1.  CLOSED                                               LISTEN

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

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

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

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

Basic 3-Way Handshake for Connection Synchronization

Figure 5

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

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

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

```
      TCP A                                                TCP B

  1.  CLOSED                                               CLOSED

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

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

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

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

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

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

Simultaneous Connection Synchronization

Figure 6

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

```
    TCP A                                                TCP B

1.  CLOSED                                               LISTEN

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

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

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

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

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

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

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

Recovery from Old Duplicate SYN

Figure 7

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

Half-Open Connections and Other Anomalies

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

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

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

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

```
      TCP A                                           TCP B

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

  2.  CLOSED                                           ESTABLISHED

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

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

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

  6.  SYN-SENT                                         CLOSED

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

Half-Open Connection Discovery

Figure 8

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

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

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

```
      TCP A                                                TCP B

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

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

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

Active Side Causes Half-Open Connection Discovery

Figure 9

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

```
    TCP A                                         TCP B

1.  LISTEN                                        LISTEN

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

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

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

5.  LISTEN                                        LISTEN
```

Old Duplicate SYN Initiates a Reset on two Passive Sockets

Figure 10

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

Reset Generation

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

There are three groups of states:

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

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

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

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

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

If an incoming segment has a security level, or compartment which 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 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 he is told that the other side has CLOSED also. Thus, a program could initiate several SENDs followed by a CLOSE, and then continue to RECEIVE until signaled that a RECEIVE failed because the other side has CLOSED. We assume that the TCP will signal a user, even if no RECEIVEs are outstanding, that the other side has closed, so the user can terminate his side gracefully. A TCP will reliably deliver all buffers SENT before the connection was CLOSED so a user who expects no data in return need only wait to hear the connection was CLOSED successfully to know that all his data was received at the destination TCP. Users must keep reading connections they close for sending until the TCP says no more data.

There are essentially three cases:

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

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

3) Both users CLOSE simultaneously

Case 1: Local user initiates the close

In this case, a FIN segment can be constructed and placed on the outgoing segment queue. No further SENDs from the user will be accepted by the TCP, and it enters the FIN-WAIT-1 state. RECEIVEs are allowed in this state. All segments preceding and including FIN will be retransmitted until acknowledged. When the other TCP

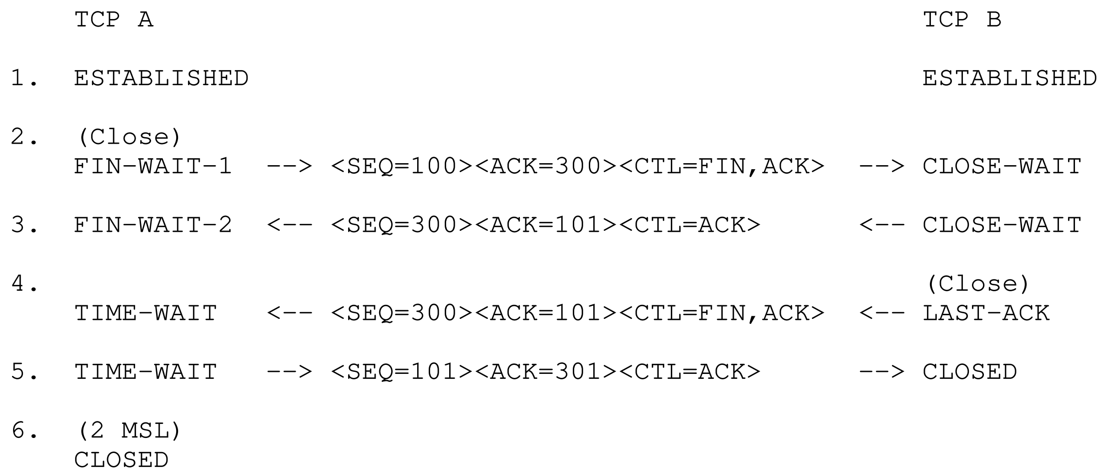
has both acknowledged the FIN and sent a FIN of its own, the first TCP can ACK this FIN. Note that a TCP receiving a FIN will ACK but not send its own FIN until its user has CLOSED the connection also.

Case 2: TCP receives a FIN from the network

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

Case 3: both users close simultaneously

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



Normal Close Sequence

Figure 11

```
    TCP A                                                TCP B

1.  ESTABLISHED                                          ESTABLISHED

2.  (Close)                                              (Close)
    FIN-WAIT-1  --> <SEQ=100><ACK=300><CTL=FIN,ACK>  ... FIN-WAIT-1
                <-- <SEQ=300><ACK=100><CTL=FIN,ACK>  <--
                ... <SEQ=100><ACK=300><CTL=FIN,ACK>  -->

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

4.  TIME-WAIT                                            TIME-WAIT
    (2 MSL)                                              (2 MSL)
    CLOSED                                               CLOSED
```

Simultaneous Close Sequence

Figure 12

A TCP connection may terminate in two ways: (1) the normal TCP close sequence using a FIN handshake, and (2) an "abort" in which one or more RST segments are sent and the connection state is immediately discarded. If the local TCP connection is closed by the remote side due to a FIN or RST received from the remote side, then the local application MUST be informed whether it closed normally or was aborted (MUST-12).

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 TCP, or if new data is received after CLOSE is called, its TCP SHOULD send a RST to show that data was lost (SHLD-3). See [17] section 2.17 for discussion.

When a connection is closed actively, it MUST linger in TIME-WAIT state for a time 2xMSL (Maximum Segment Lifetime) (MUST-13).

However, it MAY accept a new SYN from the remote TCP 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 [30] 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. Precedence and Security

The IPv4 specification [1] includes a precedence value in the (now obsoleted) Type of Service field (TOS) field. It was modified in [15], and then obsoleted by the definition of Differentiated Services (DiffServ) [6]. Setting and conveying TOS between the network layer, TCP, and applications is obsolete, and replaced by DiffServ in the current TCP specification.

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) [38].

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 [18] 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.

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 [24]. 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 on the relation between IP security.

3.7. Segmentation

The term "segmentation" refers to the activity TCP performs when ingesting a stream of bytes from a sending application and packetizing that stream of bytes into TCP segments. Individual TCP segments often do not correspond one-for-one to individual send (or socket write) calls from the application. Applications may perform writes at the granularity of messages in the upper layer protocol, but TCP guarantees no boundary coherence between the TCP segments sent and received versus user application data read or write buffer boundaries. In some specific protocols, such as RDMA using DDP and MPA [22], 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 strcutures, 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 machines 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 segments larger than the smallest MTU within an IP network path, because this results in either packet loss or fragmentation. Making matters worse, some firewalls or middleboxes may drop fragmented packets or ICMP messages related related to fragmentation.

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

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

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

3.7.1. Maximum Segment Size Option

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

TCP 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 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 really sends, the "effective send MSS," MUST be the smaller (MUST-16) of the send MSS (which reflects the available reassembly buffer size at the remote host, the EMTU_R [14]) and the largest transmission size permitted by the IP layer (EMTU_S [14]):

```
Eff.snd.MSS =

    min(SendMSS+20, MMS_S) - TCPhdrsize - IPoptionsize
```

where:

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

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

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

- o IPoptionsize is the size of any IP options associated with a TCP connection. Note that some options may not be included on all packets, but that for each segment sent, the sender should adjust the data length accordingly, within the Eff.snd.MSS.

The MSS value to be sent in an MSS option should be equal to the effective MTU minus the fixed IP and TCP headers. By ignoring both IP and TCP options when calculating the value for the MSS option, if there are any IP or TCP options to be sent in a packet, then the sender must decrease the size of the TCP data accordingly. RFC 6691 [33] 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). 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 [14].

When TCP is used in a situation where either the IP or TCP headers are not fixed, the sender must reduce the amount of TCP data in any given packet by the number of octets used by the IP and TCP options. This has been a point of confusion historically, as explained in RFC 6691, Section 3.1.

3.7.2. Path MTU Discovery

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

PMTUD for IPv4 [2] or IPv6 [3] is implemented in conjunction between TCP, IP, and ICMP protocols. It relies both on avoiding source fragmentation and setting the IPv4 DF (don't fragment) flag, the latter to inhibit on-path fragmentation. It relies on ICMP errors from routers along the path, whenever a segment is too large to traverse a link. Several adjustments to a TCP implementation with PMTUD are described in RFC 2923 in order to deal with problems experienced in practice [8]. PLPMTUD [19] 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.

3.7.3. Interfaces with Variable MTU Values

The effective MTU can sometimes vary, as when used with variable compression, e.g., RObust Header Compression (ROHC) [26]. It is tempting for TCP to want to advertise the largest possible MSS, to support the most efficient use of compressed payloads. Unfortunately, some compression schemes occasionally need to transmit full headers (and thus smaller payloads) to resynchronize state at their endpoint compressors/decompressors. If the largest MTU is used to calculate the value to advertise in the MSS option, TCP retransmission may interfere with compressor resynchronization.

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

3.7.4. Nagle Algorithm

The "Nagle algorithm" was described in RFC 896 [13] and was recommended in RFC 1122 [14] 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 buffers all user data (regardless of the PSH bit), until the outstanding data has been acknowledged or until the TCP can send a full-sized segment (Eff.snd.MSS bytes).

A TCP 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 [25].

3.7.5. IPv6 Jumbograms

In order to support TCP over IPv6 jumbograms, implementations need to be able to send TCP segments larger than the 64KB limit that the MSS option can convey. RFC 2675 [7] 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.

3.8. Data Communication

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

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

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

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

3.8.1. Retransmission Timeout

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

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

RFC 793 contains an early example procedure for computing the RTO. This was then replaced by the algorithm described in RFC 1122, and subsequently updated in RFC 2988, and then again in RFC 6298.

If a retransmitted packet is identical to the original packet (which implies not only that the data boundaries have not changed, but also that the window and acknowledgment fields of the header have not changed), then the same IP Identification field MAY be used (see Section 3.2.1.5 of RFC 1122) (MAY-4).

3.8.2. TCP Congestion Control

RFC 1122 required implementation of Van Jacobson's congestion control algorithm combining slow start with congestion avoidance. RFC 2581 provided IETF Standards Track description of this, along with fast retransmit and fast recovery. RFC 5681 is the current description of these algorithms and is the current standard for TCP congestion control.

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

Explicit Congestion Notification (ECN) was defined in RFC 3168 and is an IETF Standards Track enhancement that has many benefits [39].

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

3.8.3. TCP Connection Failures

Excessive retransmission of the same segment by TCP 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 [14] Section 3.3.1.4) 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.

(d) TCP 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. The application can close the connection (i.e., give up on the open attempt) sooner, of course.

3.8.4. TCP Keep-Alives

Implementors MAY include "keep-alives" in their TCP implementations (MAY-5), although this practice is not universally accepted. 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 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.

3.8.5. The Communication of Urgent Information

As a result of implementation differences and middlebox interactions, new applications SHOULD NOT employ the TCP urgent mechanism (SHLD-13). However, TCP implementations MUST still include support for the urgent mechanism (MUST-30). Details can be found in RFC 6093 [29].

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

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

The method employs a urgent field which is carried in all segments transmitted. The URG control flag indicates that the urgent field is meaningful and must be added to the segment sequence number to yield

the urgent pointer. The absence of this flag indicates that there is no urgent data outstanding.

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

A TCP MUST support a sequence of urgent data of any length (MUST-31). [14]

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

A TCP MUST (MUST-32) inform the application layer asynchronously whenever it receives an Urgent pointer and there was previously no pending urgent data, or whenvever the Urgent pointer advances in the data stream. There MUST (MUST-33) be a way for the application to learn how much urgent data remains to be read from the connection, or at least to determine whether or not more urgent data remains to be read. [14]

3.8.6. Managing the Window

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

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

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

Indicating a large window encourages transmissions. If more data arrives than can be accepted, it will be discarded. This will result in excessive retransmissions, adding unnecessarily to the load on the

network and the TCPs. Indicating a small window may restrict the transmission of data to the point of introducing a round trip delay between each new segment transmitted.

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

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

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

3.8.6.1. Zero Window Probing

The sending TCP must be prepared to accept from the user and send at least one octet of new data even if the send window is zero. The sending TCP must regularly retransmit to the receiving TCP even when the window is zero, 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 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 MAY keep its offered receive window closed indefinitely (MAY-8). As long as the receiving TCP continues to send acknowledgments in response to the probe segments, the sending TCP 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 [31].

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

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

3.8.6.2. Silly Window Syndrome Avoidance

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

3.8.6.2.1. Sender's Algorithm - When to Send Data

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

A TCP 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.

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 which has been found to work well is for the sender to calculate Max(SND.WND), the maximum send window it has seen so far on the connection, and to use this value as an estimate of RCV.BUFF. Unfortunately, this can only be an estimate; the receiver may at any time reduce the size of RCV.BUFF. To avoid a resulting deadlock, it is necessary to have a timeout to force transmission of data, overriding the SWS avoidance algorithm. In practice, this timeout should seldom occur.

The "useable window" is:

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

i.e., the offered window less the amount of data sent but not acknowledged. If D is the amount of data queued in the sending TCP 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 Section 3.8.6.1).

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

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

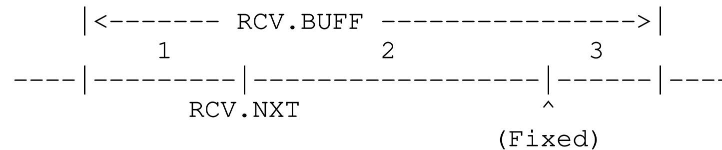
The receiver's SWS avoidance algorithm determines when the right window edge may be advanced; this is customarily known as "updating the window". This algorithm combines with the delayed ACK algorithm (see Section 3.8.6.3) to determine when an ACK segment containing the current window will really be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

The solution to receiver SWS is to avoid advancing the right window edge RCV.NXT+RCV.WND in small increments, even if data is received from the network in small segments.

Suppose the total receive buffer space is RCV.BUFF. At any given moment, RCV.USER octets of this total may be tied up with data that

has been received and acknowledged but which the user process has not yet consumed. When the connection is quiescent, RCV.WND = RCV.BUFF and RCV.USER = 0.

Keeping the right window edge fixed as data arrives and is acknowledged requires that the receiver offer less than its full buffer space, i.e., the receiver must specify a RCV.WND that keeps RCV.NXT+RCV.WND constant as RCV.NXT increases. Thus, the total buffer space RCV.BUFF is generally divided into three parts:




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

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

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

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

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

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

3.8.6.3. 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 SHOULD implement a delayed ACK (SHLD-18), but an ACK should not be excessively delayed; in particular, the delay MUST be less than 0.5 seconds (MUST-40), and in a stream of full-sized segments there SHOULD be an ACK for at least every second segment (SHLD-19). Excessive delays on ACK's can disturb the round-trip timing and packet "clocking" algorithms.

3.9. Interfaces

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

3.9.1. User/TCP Interface

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

TCP User Commands

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

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

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

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

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

Open

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

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

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

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

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 that supports multiple concurrent users 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 will begin the procedure to synchronize (i.e., establish) the connection at once.

The timeout, if present, permits the caller to set up a timeout for all data submitted to TCP. If data is not successfully delivered to the destination within the timeout period, the TCP

will abort the connection. The present global default is five minutes.

The TCP or some component of the operating system will verify the users authority to open a connection with the specified 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. The local connection name can then be used as a short hand term for the connection defined by the <local socket, foreign 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 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 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 MAY implement PUSH flags on SEND calls (MAY-15). If PUSH flags are not implemented, then the sending TCP: (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 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 may buffer the data before sending, without regard to the PUSH flag (see Section 3.7.4).

An application program is logically required to set the PUSH flag in a SEND call whenever it needs to force delivery of the

data to avoid a communication deadlock. However, a TCP SHOULD send a maximum-sized segment whenever possible (SHLD-28), to improve performance (see Section 3.8.6.2.1).

New applications SHOULD NOT set the URGENT flag [29] due to implementation differences and middlebox issues (SHLD-13).

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

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

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

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

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

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

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

Receive

Format: RECEIVE (local connection name, buffer address, byte count) -> byte count, urgent flag, push flag (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 allocate buffer storage, or the TCP might share a ring buffer with the user.

Close

Format: CLOSE (local connection name)

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

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

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

Close also implies push function.

Status

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

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

This command returns a data block containing the following information:

local socket,
foreign socket,
local connection name,
receive window,
send window,
connection state,
number of buffers awaiting acknowledgment,
number of buffers pending receipt,
urgent state,
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 TCP on the other side of the connection. Depending on the implementation, users may receive abort indications for each outstanding SEND or RECEIVE, or may simply receive an ABORT-acknowledgment.

Flush

Some TCP implementations have included a FLUSH call, which will empty the TCP send queue of any data for which the user has issued SEND calls but which 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 3.9.2.2) (TODO - MUST here is inconsistent with SHOULDs in the section describing ICMP processing)

* Excessive retransmissions (see Section 3.8.3) (TODO - MUST here is inconsistent with SHOULD in the section describing excessive retransmissions)

* Urgent pointer advance (see Section 3.8.5) (MUST-32).

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

3.9.2. TCP/Lower-Level Interface

The TCP calls on a lower level protocol module to actually send and receive information over a network. The two current standard Internet Protocol (IP) versions layered below TCP are IPv4 [1] and IPv6 [5].

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) [38]. 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 MUST ignore options that it does not understand (MUST-50).

A TCP MAY support the Time Stamp (MAY-10) and Record Route (MAY-11) options.

3.9.2.1. Source Routing

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

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

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

3.9.2.2. ICMP Messages

TCP 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.

[23] 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 MUST silently discard any received ICMP Source Quench messages (MUST-55). See [11] 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 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 SHOULD abort the connection (SHLD-26). [23] 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 [23] section 4 describes widespread implementation behavior that treats soft errors as hard errors during connection establishment.

3.9.2.3. Remote Address Validation

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

> An incoming SYN with an invalid source address MUST be ignored either by TCP or by the IP layer (MUST-63) (see Section 3.2.1.3 of [14]).
>
> A TCP implementation MUST silently discard an incoming SYN segment that is addressed to a broadcast or multicast address (MUST-57).

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

3.10. Event Processing

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

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

Events that occur:

User Calls

- OPEN
- SEND
- RECEIVE
- CLOSE
- ABORT
- STATUS

Arriving Segments

- SEGMENT ARRIVES

Timeouts

- USER TIMEOUT
- RETRANSMISSION TIMEOUT
- TIME-WAIT TIMEOUT

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

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

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

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

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

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

OPEN Call

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

Create a new transmission control block (TCB) to hold connection state information. Fill in local socket identifier, foreign socket, DiffServ field, security/compartment, and user timeout information. Note that some parts of the foreign socket may be unspecified in a passive OPEN and are to be filled in by the parameters of the incoming SYN segment. Verify the security and 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 foreign socket is unspecified, return "error: foreign socket unspecified"; if active and the foreign socket is specified, issue a SYN segment. An initial send sequence number (ISS) is selected. A SYN segment of the form <SEQ=ISS><CTL=SYN> is sent. Set SND.UNA to ISS, SND.NXT to ISS+1, enter SYN-SENT state, and return.

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

LISTEN STATE

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

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

Return "error: connection already exists".

SEND Call

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

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

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

LISTEN STATE

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

SYN-SENT STATE
SYN-RECEIVED STATE

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

ESTABLISHED STATE
CLOSE-WAIT STATE

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

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

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

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

RECEIVE Call

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

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

Otherwise return "error: connection does not exist".

LISTEN STATE
SYN-SENT STATE
SYN-RECEIVED STATE

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

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

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

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

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

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

CLOSE-WAIT STATE

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

CLOSING STATE
LAST-ACK STATE

TIME-WAIT STATE

Return "error: connection closing".

CLOSE Call

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

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

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

LISTEN STATE

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

SYN-SENT STATE

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

SYN-RECEIVED STATE

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

ESTABLISHED STATE

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

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

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

CLOSE-WAIT STATE

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

CLOSING STATE
LAST-ACK STATE
TIME-WAIT STATE

Respond with "error: connection closing".

ABORT Call

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

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

Otherwise return "error: connection does not exist".

LISTEN STATE

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

SYN-SENT STATE

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

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

Send a reset segment:

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

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

CLOSING STATE LAST-ACK STATE TIME-WAIT STATE

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

STATUS Call

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

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

Otherwise return "error: connection does not exist".

LISTEN STATE

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

SYN-SENT STATE

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

SYN-RECEIVED STATE

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

ESTABLISHED STATE

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

FIN-WAIT-1 STATE

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

FIN-WAIT-2 STATE

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

CLOSE-WAIT STATE

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

CLOSING STATE

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

LAST-ACK STATE

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

TIME-WAIT STATE

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

SEGMENT ARRIVES

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

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

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

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

If the ACK bit is on,

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

Return.

If the state is LISTEN then

first check for an RST

An incoming RST should be ignored. Return.

second check for an ACK

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

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

Return.

third check for a SYN

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

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

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

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

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

fourth other text or control

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

If the state is SYN-SENT then

first check the ACK bit

If the ACK bit is set

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

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

and discard the segment. Return.

If SND.UNA < SEG.ACK =< SND.NXT then the ACK is acceptable. 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 peer TCP 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 [28], with the mitigation that a TCP implementation SHOULD first check that the sequence number exactly matches RCV.NXT prior to executing the action in the next paragraph.

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

third check the security

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

If there is an ACK

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

Otherwise

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

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

fourth check the SYN bit

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

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

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

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

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

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

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

and send it. Set the variables:

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

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

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 [16] and is used in TCP Fast Open (TFO) [36], 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 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 [30] 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 section 3 describes a potential blind reset attack and optional mitigation approach that SHOULD be implemented. For stacks implementing RFC 5961, the three checks below apply, otherwise processesing 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 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 MUST send an acknowledgement (challenge ACK):

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

After sending the challenge ACK, TCP 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 ports

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 [28]. For the TIME-WAIT state, new connections can be accepted if the timestamp option is used and meets expectations (per [30]). For all other caess, RFC 5961 provides a mitigation that SHOULD be implemented, though there are alternatives (see Section 6). RFC 5961 recommends that in these synchronized states, if the SYN bit is set, irrespective of the sequence number, TCP MUST send a "challenge ACK" to the remote peer:

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

After sending the acknowledgement, TCP 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 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

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

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

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

FIN-WAIT-1 STATE

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

FIN-WAIT-2 STATE

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

CLOSE-WAIT STATE

Do the same processing as for the ESTABLISHED state.

CLOSING STATE

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

LAST-ACK STATE

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

TIME-WAIT STATE

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

sixth, check the URG bit,

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

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

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

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

seventh, process the segment text,

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

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

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

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

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

Please note the window management suggestions in Section 3.8.

Send an acknowledgment of the form:

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

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

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

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

eighth, check the FIN bit,

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

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

SYN-RECEIVED STATE
ESTABLISHED STATE

Enter the CLOSE-WAIT state.

FIN-WAIT-1 STATE

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

FIN-WAIT-2 STATE

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

CLOSE-WAIT STATE

Remain in the CLOSE-WAIT state.

CLOSING STATE

Remain in the CLOSING state.

LAST-ACK STATE

Remain in the LAST-ACK state.

TIME-WAIT STATE

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

and return.

USER TIMEOUT

USER TIMEOUT

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

RETRANSMISSION TIMEOUT

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

TIME-WAIT TIMEOUT

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

3.11. Glossary

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

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

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

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

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

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

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

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

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

FTP
: A file transfer protocol.

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

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

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

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

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

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

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

IP
Internet Protocol.

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

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

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

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

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

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

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

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

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

octet
: An eight bit byte.

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

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

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

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

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

RCV.NXT
: receive next sequence number

RCV.UP
: receive urgent pointer

RCV.WND
: receive window

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

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

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

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

SEG.ACK
segment acknowledgment

SEG.LEN
segment length

SEG.SEQ
segment sequence

SEG.UP
segment urgent pointer field

SEG.WND
segment window field

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

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

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

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

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

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

SND.NXT
: send sequence

SND.UNA
: left sequence

SND.UP
: send urgent pointer

SND.WL1
: segment sequence number at last window update

SND.WL2
: segment acknowledgment number at last window update

SND.WND
: send window

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

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

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

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

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 [6] containing the Differentiated Services Code Point (DSCP) value and two unused bits.

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

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

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

4. Changes from RFC 793

This document obsoletes RFC 793 as well as RFC 6093 and 6528, which updated 793. In all cases, only the normative protocol specification and requirements have been incorporated into this document, and the informational text with background and rationale has not been carried in. The informational content of those documents is still valuable in learning about and understanding TCP, and they are valid Informational references, even though their normative content has been incorporated into this document.

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

The collection of applicable RFC Errata that have been reported and either accepted or held for an update to RFC 793 were incorporated (Errata IDs: 573, 574, 700, 701, 1283, 1561, 1562, 1564, 1565, 1571, 1572, 2296, 2297, 2298, 2748, 2749, 2934, 3213, 3300, 3301). Some errata were not applicable due to other changes (Errata IDs: 572, 575, 1569, 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 mangement concerns allow connection resources to be reclaimed. RFC 6429 is obsoleted in the sense that this clarification has been reflected in this update to the base TCP specification now.

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

This document started as draft-eddy-rfc793bis-00, that was merely a proposal and rough plan for updating RFC 793.

The -01 revision of this draft-eddy-rfc793bis incorporates the content of RFC 793 Section 3 titled "FUNCTIONAL SPECIFICATION". Other content from RFC 793 has not been incorporated. The -01 revision of this document makes some minor formatting changes to the RFC 793 content in order to convert the content into XML2RFC format and account for left-out parts of RFC 793. For instance, figure numbering differs and some indentation is not exactly the same.

The -02 revision of draft-eddy-rfc793bis incorporates errata that have been verified:

Errata ID 573: Reported by Bob Braden (note: This errata basically is just a reminder that RFC 1122 updates 793. Some of the associated changes are left pending to a separate revision that incorporates 1122. Bob's mention of PUSH in 793 section 2.8 was not applicable here because that section was not part of the "functional specification". Also the 1122 text on the retransmission timeout also has been updated by subsequent RFCs, so the change here deviates from Bob's suggestion to apply the 1122 text.)
Errata ID 574: Reported by Yin Shuming
Errata ID 700: Reported by Yin Shuming
Errata ID 701: Reported by Yin Shuming
Errata ID 1283: Reported by Pei-chun Cheng
Errata ID 1561: Reported by Constantin Hagemeier
Errata ID 1562: Reported by Constantin Hagemeier
Errata ID 1564: Reported by Constantin Hagemeier
Errata ID 1565: Reported by Constantin Hagemeier
Errata ID 1571: Reported by Constantin Hagemeier
Errata ID 1572: Reported by Constantin Hagemeier
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 pseudoheader

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 formating 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 implemente 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.

Some other suggested changes that will not be incorporated in this 793 update unless TCPM consensus changes with regard to scope are:

1. look at Tony Sabatini suggestion for describing DO field
2. per discussion with Joe Touch (TAPS list, 6/20/2015), the description of the API could be revisited

Early in the process of updating RFC 793, Scott Brim mentioned that this should include a PERPASS/privacy review. This may be something for the chairs or AD to request during WGLC or IETF LC.

5. IANA Considerations

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

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

6. Security and Privacy Considerations

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. [28]) 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 [28]). 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) [27] 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) or none at all (e.g. TLS). Other security features that have been added to TCP (e.g. ISN generation, sequence number checks, etc.) 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 [21]. 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" [44]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 [37] 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 [29] 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) that act as mitigations to these attacks.

Other attacks are focused on exhausting the resources of a TCP server. Examples include SYN flooding [20] or wasting resources on non-progressing connections [31]. 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:

Michael Scharf
Yoshifumi Nishida
Pasi Sarolahti

During early discussion of this work on the TCPM mailing list, and at the IETF 88 meeting in Vancouver, helpful comments, critiques, and reviews were received from (listed alphebetically): David Borman, Yuchung Cheng, Martin Duke, Kevin Lahey, Kevin Mason, Matt Mathis, Hagen Paul Pfeifer, Anthony Sabatini, Joe Touch, Reji Varghese, Lloyd Wood, and Alex Zimmermann. Joe Touch provided help in clarifying the description of segment size parameters and PMTUD/PLPMTUD recommendations.

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

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

[6] 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>.

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

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

[11] Gont, F., "Deprecation of ICMP Source Quench Messages", RFC 6633, DOI 10.17487/RFC6633, May 2012, <https://www.rfc-editor.org/info/rfc6633>.

8.2. Informative References

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

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

[14] 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>.

[15] 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>.

[16] 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>.

[17] 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>.

[18] 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>.

[19] 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>.

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

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

[22] 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>.

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

[24] 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>.

[25] 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>.

[26] 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>.

[27] 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>.

[28] 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>.

[29] 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>.

[30] 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>.

[31] 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>.

[32] 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>.

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

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

[35] 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>.

[36] 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>.

[37] 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>.

[38] 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>.

[39] 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>.

[40] 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>.

[41] IANA, "Transmission Control Protocol (TCP) Parameters, https://www.iana.org/assignments/tcp-parameters/ tcp-parameters.xhtml", 2017.

[42] Gont, F., "Processing of IP Security/Compartment and Precedence Information by TCP", draft-gont-tcpm-tcp-seccomp-prec-00 (work in progress), March 2012.

[43] Gont, F. and D. Borman, "On the Validation of TCP Sequence Numbers", draft-gont-tcpm-tcp-seq-validation-02 (work in progress), March 2015.

[44] 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.

[45] Minshall, G., "A Proposed Modification to Nagle's Algorithm", draft-minshall-nagle-01 (work in progress), June 1999.

[46] Dalal, Y. and C. Sunshine, "Connection Management in Transport Protocols", Computer Networks Vol. 2, No. 6, pp. 454-473, December 1978.

Appendix A. Other Implementation Notes

This section includes additional notes and references on TCP implementation decisions that are currently not a part of the RFC series or included within the TCP standard. These items can be considered by implementers, but there was not yet a consensus to include them in the standard.

A.1. IP Security Compartment and Precedence

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 [18] 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.

Implementers of MLS systems that use IP security options (e.g. IPSO, CIPSO, or CALIPSO) should implement any additional logic appropriate for their requirements.

Reseting connections when incoming packets do not meet expected security compartment or precedence expectations has been recognized as a possible attack vector [42], and there has been discussion about ammending the TCP specification to prevent connections from being aborted due to non-matching IP security compartment and DiffServ codepoint values.

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 [43], 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 occuring. 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 [43].

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 [45] 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

TODO - mention the low watermark function that is in Linux - suggested by Michael Welzl

SO_SNDLOWAT and SO_RCVLOWAT would be potential enhancements to the abstract TCP API

TCP_NOTSENT_LOWAT is what Michael is talking about, that helps a sending TCP application to help avoid creating large amounts of buffered data (and corresponding latency). This is useful for applications that are multiplexing data from multiple upper level streams onto a connection, especially when streams may be a mix of interactive/realtime 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.

```
                                                  |        | | | |S| |
                                                  |        | | | |H| |F
                                                  |        | | | |O|M|o
                                                  |        | |S| |U|U|o
                                                  |        | |H| |L|S|t
                                                  |        |M|O| |D|T|n
                                                  |        |U|U|M| | |o
                                                  |        |S|L|A|N|N|t
```

FEATURE	ReqID	T	D	Y	O T	O T	t e
Push flag							
Aggregate or queue un-pushed data	MAY-16			x			
Sender collapse successive PSH flags	SHLD-27		x				
SEND call can specify PUSH	MAY-15			x			
If cannot: sender buffer indefinitely	MUST-60					x	
If cannot: PSH last segment	MUST-61	x					
Notify receiving ALP of PSH	MAY-17			x			1
Send max size segment when possible	SHLD-28		x				
Window							
Treat as unsigned number	MUST-1	x					
Handle as 32-bit number	REC-1		x				
Shrink window from right	SHLD-14				x		
- Send new data when window shrinks	SHLD-15				x		
- Retransmit old unacked data within window	SHLD-16		x				
- Time out conn for data past right edge	SHLD-17				x		
Robust against shrinking window	MUST-34	x					
Receiver's window closed indefinitely	MAY-8			x			
Use standard probing logic	MUST-35	x					
Sender probe zero window	MUST-36	x					
First probe after RTO	SHLD-29		x				
Exponential backoff	SHLD-30		x				
Allow window stay zero indefinitely	MUST-37	x					
Retransmit old data beyond SND.UNA+SND.WND	MAY-7			x			
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					
Cope with illegal option length	MUST-7	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			
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 RFC 5681	MUST-19	x				
Retransmit with same IP ident	MAY-4			x		
Karn's algorithm	MUST-18	x				
Generating ACK's:						
Aggregate whenever possible	MUST-58	x				
Queue out-of-order segments	SHLD-31		x			
Process all Q'd before send ACK	MUST-59	x				
Send ACK for out-of-order segment	MAY-13			x		
Delayed ACK's	SHLD-18		x			
Delay < 0.5 seconds	MUST-40	x				
Every 2nd full-sized segment ACK'd	SHLD-19	x				
Receiver SWS-Avoidance Algorithm	MUST-39	x				
Sending data						
Configurable TTL	MUST-49	x				

Sender SWS-Avoidance Algorithm	MUST-38	x					
Nagle algorithm	SHLD-7		x				
Application can disable Nagle algorithm	MUST-17	x					
Connection Failures:							
Negative advice to IP on R1 retxs	MUST-20	x					
Close connection on R2 retxs	MUST-20	x					
ALP can set R2	MUST-21	x					1
Inform ALP of R1<=retxs<R2	SHLD-9		x				1
Recommended value for R1	SHLD-10		x				
Recommended value for R2	SHLD-11		x				
Same mechanism for SYNs	MUST-22	x					
R2 at least 3 minutes for SYN	MUST-23	x					
Send Keep-alive Packets:	MAY-5			x			
- Application can request	MUST-24	x					
- Default is "off"	MUST-25	x					
- Only send if idle for interval	MUST-26	x					
- Interval configurable	MUST-27	x					
- Default at least 2 hrs.	MUST-28	x					
- Tolerant of lost ACK's	MUST-29	x					
- Send with no data	SHLD-12		x				
- Configurable to send garbage octet	MAY-6			x			
IP Options							
Ignore options TCP doesn't understand	MUST-50	x					
Time Stamp support	MAY-10			x			
Record Route support	MAY-11			x			
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							

Feature	ReqID						
Error Report mechanism	MUST-47	x					
ALP can disable Error Report Routine	SHLD-20		x				
ALP can specify DiffServ field for sending	MUST-48	x					
Passed unchanged to IP	SHLD-22		x				
ALP can change DiffServ field during connection	SHLD-21		x				
ALP generally changing DiffServ during conn.	SHLD-23				x		
Pass received DiffServ field up to ALP	MAY-9			x			
FLUSH call	MAY-14			x			
Optional local IP addr parm. in OPEN	MUST-43	x					
RFC 5961 Support:							
Implement data injection protection	MAY-12			x			
Explicit Congestion Notification:							
Support ECN	SHLD-8		x				

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

Author's Address

Wesley M. Eddy (editor)
MTI Systems
US

Email: wes@mti-systems.com

TCPM WG
Internet Draft
Updates: 793
Intended status: Standards Track
Expires: January 2019
July 19, 2018

J. Touch

Wes Eddy
MTI Systems

TCP Extended Data Offset Option
draft-ietf-tcpm-tcp-edo-10.txt

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), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html

This Internet-Draft will expire on January 19, 2019.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in

Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Abstract

TCP segments include a Data Offset field to indicate space for TCP options but the size of the field can limit the space available for complex options such as SACK and Multipath TCP and can limit the combination of such options supported in a single connection. This document updates RFC 793 with an optional TCP extension to that space to support the use of multiple large options. It also explains why the initial SYN of a connection cannot be extending a single segment.

Table of Contents

1. Introduction...3
2. Conventions used in this document..............................3
3. Motivation...3
4. Requirements for Extending TCP’s Data Offset...................4
5. The TCP EDO Option...4
   5.1. EDO Supported...5
   5.2. EDO Extension...5
   5.3. The two EDO Extension variants............................8
6. TCP EDO Interaction with TCP...................................9
   6.1. TCP User Interface..9
   6.2. TCP States and Transitions................................9
   6.3. TCP Segment Processing...................................10
   6.4. Impact on TCP Header Size................................10
   6.5. Connectionless Resets....................................11
   6.6. ICMP Handling..11
7. Interactions with Middleboxes.................................12
   7.1. Middlebox Coexistence with EDO...........................12
   7.2. Middlebox Interference with EDO..........................13
8. Comparison to Previous Proposals..............................14
   8.1. EDO Criteria...14
   8.2. Summary of Approaches....................................15
   8.3. Extended Segments..16
   8.4. TCPx2..16
   8.5. LO/SLO...17
   8.6. LOIC...17
   8.7. Problems with Extending the Initial SYN..................18
9. Implementation Issues...19
10. Security Considerations......................................20
11. IANA Considerations..20
12. References...20

12.1. Normative References....................................20
12.2. Informative References..................................20
13. Acknowledgments...22

1. Introduction

TCP's Data Offset (DO)is a 4-bit field, which indicates the number of 32-bit words of the entire TCP header [RFC793]. This limits the current total header size to 60 bytes, of which the basic header occupies 20, leaving 40 bytes for options. These 40 bytes are increasingly becoming a limitation to the development of advanced capabilities, such as when SACK [RFC2018][RFC6675] is combined with either Multipath TCP [RFC6824], TCP-AO [RFC5925], or TCP Fast Open [RFC7413].

This document specifies the TCP Extended Data Offset (EDO) option, and is independent of (and thus compatible with) IPv4 and IPv6. EDO extends the space available for TCP options, except for the initial SYN and SYN/ACK. This document also explains why the option space of the initial SYN segments cannot be extended as individual segments without severe impact on TCP's initial handshake and the SYN/ACK limitation that results from potential middlebox misbehavior. Multiple other TCP extensions are being considered in the TCPM working group in order to address the case of SYN and SYN/ACK segments [Bo14][Br14][To18]. Some of these other extensions can work in conjunction with EDO (e.g., [To18]).

2. Conventions used in this document

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

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

In this document, the characters ">>" preceding an indented line(s) indicates a compliance requirement statement using the key words listed above. This convention aids reviewers in quickly identifying or finding the explicit compliance requirements of this RFC.

3. Motivation

TCP supports headers with a total length of up to 15 32-bit words, as indicated in the 4-bit Data Offset field [RFC793]. This accounts

for a total of 60 bytes, of which the default TCP header fields occupy 20 bytes, leaving 40 bytes for options.

TCP connections already use this option space for a variety of capabilities. These include Maximum Segment Size (MSS) [RFC793], Window Scale (WS) [RFC7323], Timestamp (TS) [RFC7323], Selective Acknowledgement (SACK) [RFC2018][RFC6675], TCP Authentication Option (TCP-AO) [RFC5925], Multipath TCP (MP-TCP)_[RFC6824], and TCP User Timeout [RFC5482]. Some options occur only in a SYN or SYN/ACK (MSS, WS), and others vary in size when used in SYN vs. non-SYN segments.

Each of these options consumes space, where some options consuming as much space as available (SACK) and other desired combinations can easily exceed the currently available space. For example, it is not currently possible to use TCP-AO with both TS and MP-TCP in the same non-SYN segment, i.e., to combine accurate round-trip estimation, authentication, and multipath support in the same connection - even though these options can be negotiated during a SYN exchange (10 for TS, 16 for TCP-AO, and 12 for MP-TCP).

TCP EDO is intended to overcome this limitation for non-SYN segments, as well as to increase the space available for SACK blocks. Further discussion of the impact of EDO and existing options is discussed in Section 6.4. Extending SYN segments is much more complicated, as discussed in Section 8.7.

4. Requirements for Extending TCP’s Data Offset

The primary goal of extending the TCP Data Offset field is to increase the space available for TCP options in all segments except the initial SYN.

An important requirement of any such extension is that it not impact legacy endpoints. Endpoints seeking to use this new option should not incur additional delay or segment exchanges to connect to either new endpoints supporting this option or legacy endpoints without this option. We call this a "backward downgrade" capability.

An additional consideration of this extension is avoiding user data corruption in the presence of popular network devices, including middleboxes. Consideration of middlebox misbehavior can also interfere with extension in the SYN/ACK.

5. The TCP EDO Option

TCP EDO extends the option space for all segments except the initial SYN (i.e., SYN set and ACK not set) and SYN/ACK response. EDO is

indicated by the TCP option codepoint of EDO-OPT and has two types: EDO Supported and EDO Extension, as discussed in the following subsections.

5.1. EDO Supported

EDO capability is determined in both directions using a single exchange of the EDO Supported option (Figure 1). When EDO is desired on a given connection, the SYN and SYN/ACK segments include the EDO Supported option, which consists of the two required TCP option fields: Kind and Length. The EDO Supported option is used only in the SYN and SYN/ACK segments and only to confirm support for EDO in subsequent segments.

```
+--------+--------+
|  Kind  | Length |
+--------+--------+
```

Figure 1 TCP EDO Supported option

An endpoint seeking to enable EDO includes the EDO Supported option in the initial SYN. If receiver of that SYN agrees to use EDO, it responds with the EDO Supported option in the SYN/ACK. The EDO Supported option does not extend the TCP option space.

>> Connections using EDO MUST negotiate its availability during the SYN exchange of the initial three-way handshake.

>> An endpoint confirming and agreeing to EDO use MUST respond with the EDO Supported option in its SYN/ACK.

The SYN/ACK uses only the EDO Supported option (and not the EDO Extension option, below) because it may not yet be safe to extend the option space in the reverse direction due to potential middlebox misbehavior (see Section 7.2). Extension of the SYN and SYN/ACK space is addressed as a separate option (see Section 8.7).

5.2. EDO Extension

When EDO is successfully negotiated, all other segments use the EDO Extension option, of which there are two variants (Figure 2 and Figure 3). Both variants are considered equivalent and either variant can be used in any segment where the EDO Extension option is required. Both variants add a Header_Length field (in network-standard byte order), indicating the length of the entire TCP header in 32-bit words. Figure 3 depicts the longer variant, which includes an additional Segment_Length field, which is identical to the TCP

pseudoheader TCP Length field and used to detect when segments have been altered in ways that would interfere with EDO (discussed further in Section 5.3).

```
+--------+--------+--------+--------+
|  Kind  | Length |  Header_Length  |
+--------+--------+--------+--------+
```

Figure 2 TCP EDO Extension option - simple variant

```
+--------+--------+--------+--------+
|  Kind  | Length |  Header_Length  |
+--------+--------+--------+--------+
|  Segment_Length |
+--------+--------+
```

Figure 3 TCP EDO Extension option - with segment length verification

>> Once enabled on a connection, all segments in both directions MUST include the EDO Extension option. Segments not needing extension MUST set the EDO Extension option Header Length field equal to the Data Offset length.

>> The EDO Extension option MAY be used only if confirmed when the connection transitions to the ESTABLISHED state, e.g., a client is enabled after receiving the EDO Supported option in the SYN/ACK and the server is enabled after seeing the EDO Extension option in the final ACK of the three-way handshake. If either of those segments lacks the appropriate EDO option, the connection MUST NOT use any EDO options on any other segments.

Internet paths may vary after connection establishment, introducing misbehaving middleboxes (see Section 7.2). Using EDO on all segments in both directions allows this condition to be detected.

>> The EDO Supported option MAY occur in an initial SYN as desired (e.g., as expressed by the user/application) and in the SYN/ACK as confirmation, but MUST NOT be inserted in other segments. If the EDO Supported option is received in other segments, it MUST be silently ignored.

>> If EDO has not been negotiated and agreed, the EDO Extension option MUST be silently ignored on subsequent segments. The EDO Extension option MUST NOT be sent in an initial SYN segment or SYN/ACK, and MUST be silently ignored and not acknowledged if so received.

>> If EDO has been negotiated, any subsequent segments arriving without the EDO Extension option MUST be silently ignored. Such events MAY be logged as warning errors and logging MUST be rate limited.

When processing a segment, EDO needs to be visible within the area indicated by the Data Offset field, so that processing can use the EDO Header_length to override the field for that segment.

>> The EDO Extension option MUST occur within the space indicated by the TCP Data Offset.

>> The EDO Extension option indicates the total length of the header. The EDO Header_length field MUST NOT exceed that of the total segment size (i.e., TCP Length).

>> The EDO Header Length MUST be at least as large as the TCP Data Offset field of the segment in which they both appear. When the EDO Header Length equals the Data Offset length, the EDO Extension option is present but it does not extend the option space. When the EDO Header Length is invalid, the TCP segment MUST be silently dropped.

>> The EDO Supported option SHOULD be aligned on a 16-bit boundary and the EDO Extension option SHOULD be aligned on a 32-bit boundary, in both cases for simpler processing.

For example, a segment with only EDO would have a Data Offset of 6 or 7 (depending on the EDO Extension variant used), where EDO would be the first option processed, at which point the EDO Extension option would override the Data Offset and processing would continue until the end of the TCP header as indicated by the EDO Header_length field.

There are cases where it might be useful to process other options before EDO, notably those that determine whether the TCP header is valid, such as authentication, encryption, or alternate checksums. In those cases, the EDO Extension option is preferably the first option after a validation option, and the payload after the Data Offset is treated as user data for the purposes of validation.

>> The EDO Extension option SHOULD occur as early as possible, either first or just after any authentication or encryption, and SHOULD be the last option covered by the Data Offset value.

Other options are generally handled in the same manner as when the EDO option is not active, unless they interact with other options.

One such example is TCP-AO [RFC5925], which optionally ignores the contents of TCP options, so it would need to be aware of EDO to operate correctly when options are excluded from the HMAC calculation.

>> Options that depend on other options, such as TCP-AO [RFC5925] (which may include or exclude options in MAC calculations) MUST also be augmented to interpret the EDO Extension option to operate correctly.

5.3. The two EDO Extension variants

There are two variants of the EDO Extension option; one includes a copy of the TCP segment length, copied from the TCP pseduoheader [RFC793]. The Segment_Length field is added to the longer variant to detect when segments are incorrectly and inappropriately merged by middleboxes or TCP offload processing but without consideration for the additional option space indicated by the EDO Header_Length field. Such effects are described in further detail in Section 7.2.

>> An endpoint MAY use either variant of the EDO Extension option interchangeably.

When the longer, 6-byte variant is used, the Segment_Length field is used to check whether modification of the segment was performed consistent with knowledge of the EDO option. The Segment_Length field will detect any modification of the length of the segment, such as might occur when segments are split or merged, that occurs without also updating the Segment Length field as well. The Segment Length field thus helps endpoints detects devices that merge or split TCP segments without support for EDO. Devices that merge or split TCP segments that support EDO would update the Segment Length field as needed, but would also ensure that the user data is handled separately from the extended option space indicate by EDO.

>> When an endpoint creates a new segment using the 6-byte EDO Extension option, the Segment_Length field is initialized with a copy of the segment length from the TCP pseudoheader.

>> When an endpoint receives a segment using the 6-byte EDO Extension option, it MUST validate the Segment_Length field with the length of the segment as indicated in the TCP pseudoheader. If the segment lengths do not match, the segment MUST be discarded and an error SHOULD be logged in a rate-limited manner.

>> The 6-byte EDO Extension variant SHOULD be used where middlebox or TCP offload support could merge or split TCP segments without

consideration for the EDO option. Because these conditions could occur at either endpoint or along the network path, the 6-byte variant SHOULD be preferred until sufficient evidence for safe use of the 4-byte variant is determined by the community.

The field will not detect other modification of the TCP user data; such modifications would need more complex detection mechanisms, such as checksums or hashes. When these are used, as with IPsec or TCP-AO, the 4-byte variant is sufficient.

>> The 4-byte EDO Extension variant is sufficient when EDO is used in conjunction with other mechanisms that provide integrity protection, such as IPsec or TCP-AO.

6. TCP EDO Interaction with TCP

The following subsections describe how EDO interacts with the TCP specification [RFC793].

6.1. TCP User Interface

The TCP EDO option is enabled on a connection using a mechanism similar to any other per-connection option. In Unix systems, this is typically performed using the 'setsockopt' system call.

>> Implementations can also employ system-wide defaults, however systems SHOULD NOT activate this extension by default to avoid interfering with legacy applications.

>> Due to the potential impacts of legacy middleboxes (discussed in Section 7), a TCP implementation supporting EDO SHOULD log any events within an EDO connection when options that are malformed or show other evidence of tampering arrive. An operating system MAY choose to cache the list of destination endpoints where this has occurred with and block use of EDO on future connections to those endpoints, but this cache MUST be accessible to users/applications on the host. Note that such endpoint assumptions can vary in the presence of load balancers where server implementations vary behind such balancers.

6.2. TCP States and Transitions

TCP EDO does not alter the existing TCP state or state transition mechanisms.

6.3. TCP Segment Processing

TCP EDO alters segment processing during the TCP option processing step. Once detected, the TCP EDO Extension option overrides the TCP Data Offset field for all subsequent option processing. Option processing continues at the next option (if present) after the EDO Extension option.

6.4. Impact on TCP Header Size

The TCP EDO Supported option increases SYN header length by a minimum of 2 bytes, but could increase it by more depending on 32-bit word alignment. Currently popular SYN options total 19 bytes, which leaves more than enough room for the EDO Supported option:

- o SACK permitted (2 bytes in SYN, optionally 2 + 8N bytes after) [RFC2018][RFC6675]
- o Timestamp (10 bytes) [RFC7323]
- o Window scale (3 bytes) [RFC7323]
- o MSS option (4 bytes) [RFC793]

Adding the EDO Supported option would result in a total of 21 bytes of SYN option space.

Subsequent segments would use 10 bytes of option space without any SACK blocks (TS only; WS and MSS are used only in SYN and SYN/ACK) or allow up to 3 SACK blocks before needing to use EDO; with EDO, the number of SACK blocks or additional options would be substantially increased. There are also other options that are emerging in the SYN, including TCP Fast Open, which uses another 6-18 (typically 10) bytes in the SYN/ACK of the first connection and in the SYN of subsequent connections [RFC7413].

TCP EDO can also be negotiated in SYNs with either of the following large options:

- o TCP-AO (authentication) (16 bytes) [RFC5925]
- o Multipath TCP (12 bytes in SYN and SYN/ACK, 20 after) [RFC6824]

Including TCP-AO with TS, WS, SACK increases the SYN option space use to 35 bytes; with Multipath TCP the use is 31 bytes. When Multipath TCP is enabled with the typical options, later segments would require 30 bytes without SACK, thus limiting the SACK option

to one block unless EDO is also supported on at least non-SYN segments.

The full combination of the above options (47 bytes for TS, WS, MSS, SACK, TCP-AO, and MPTCP) does not fit in the existing SYN option space and (as noted) that space cannot be extended within a single SYN segment. There has been a proposal to change TS to a 2 byte "TS permitted" signal in the initial SYN, provided it can be safely enabled during the connection later or might be avoided completely [Ni15]. Even using "TS-permitted", the total space is still too large to support in the initial SYN without SYN option space extension [Bo14][Br14][To18].

The EDO Extension option has negligible impact on other headers, because it can either come first or just after security information, and in either case the additional 4 or 6 bytes are easily accommodated within the TCP Data Offset length. Once the EDO option is processed, the entirety of the remainder of the TCP segment is available for any remaining options.

6.5. Connectionless Resets

A RST may arrive during a currently active connection or may be needed to cleanup old state from an abandoned connection. The latter occurs when a new SYN is sent to an endpoint with matching existing connection state, at which point that endpoint responds with a RST and both ends remove stale information.

The EDO Extension option is mandatory on all TCP segments once negotiated, i.e., except in the SYN and SYN/ACK (which establish support) and the RST. A RST may lack the context to know that EDO is active on a connection.

>> The EDO Extension option MAY occur in a RST when the endpoint has connection state that has negotiated EDO. However, unless the RST is generated by an incoming segment that includes an EDO Extension option, the transmitted RST MUST NOT include the EDO Extension option.

6.6. ICMP Handling

ICMP responses are intended to include the IP and the port fields of TCP and UDP headers of typical TCP/IP and UDP/IP packets [RFC792]. This includes the first 8 data bytes of the original datagram, intended to include the transport port numbers used for connection demultiplexing. Later specifications encourage returning as much of the original payload as possible [RFC1812]. In either case, legacy

options or new options in the EDO extension area might or might not be included, and so options are generally not assumed to be part of ICMP processing anyway.

7. Interactions with Middleboxes

Middleboxes are on-path devices that typically examine or modify packets in ways that Internet routers do not [RFC3234]. This includes parsing transport headers and/or rewriting transport segments in ways that may affect EDO.

There are several cases to consider:

- Typical NAT/NAPT devices, which modify only IP address and/or TCP port number fields (with associated TCP checksum updates)

- Middleboxes that try to reconstitute TCP data streams, such as for deep-packet inspection for virus scanning

- Middleboxes that modify known TCP header fields

- Middleboxes that rewrite TCP segments

7.1. Middlebox Coexistence with EDO

Middleboxes can coexist with EDO when they either support EDO or when they ignore its impact on segment structure.

NATs and NAPTs, which rewrite IP address and/or transport port fields, are the most common form of middlebox and are not affected by the EDO option.

Middleboxes that support EDO would be those that correctly parse the EDO option. Such boxes can reconstitute the TCP data stream correctly or can modify header fields and/or rewrite segments without impact to EDO.

Conventional TCP proxies terminate the TCP connection in both directions and thus operate as TCP endpoints, such as when a client-middlebox and middlebox-server each have separate TCP connections. They would support EDO by following the host requirements herein on both connections. The use of EDO on one connection is independent of its use on the other in this case.

7.2. Middlebox Interference with EDO

Middleboxes that do not support EDO cannot coexist with its use when they modify segment boundaries or do not forward unknown (e.g., the EDO) options.

So-called "transparent" rewriting proxies, which inappropriately and incorrectly modify TCP segment boundaries, might mix option information with user data if they did not support EDO. Such devices might also interfere with other TCP options such as TCP-AO. There are three types of such boxes:

- o Those that process received options and transmit sent options separately, i.e., although they rewrite segments, they behave as TCP endpoints in both directions.

- o Those that split segments, taking a received segment and emitting two or more segments with revised headers.

- o Those that join segments, receiving multiple segments and emitting a single segment whose data is the concatenation of the components.

In all three cases, EDO is either treated as independent on different sides of such boxes or not. If independent, EDO would either be correctly terminated in either or both directions or disabled due to lack of SYN/ACK confirmation in either or both directions. Problems would occur only when TCP segments with EDO are combined or split while ignoring the EDO option. In the split case, the key concern is if the split happens within the option extension space or if EDO is silently copied to both segments without copying the corresponding extended option space contents. However, the most comprehensive study of these cases indicates that "although middleboxes do split and coalesce segments, none did so while passing unknown options" [Ho11].

Note that the second and third types of middlebox behaviors listed above may create syndromes similar to TCP transmit and receive hardware offload engines that incorrectly modify segments with unknown options.

Middleboxes that silently remove options that they do not implement have been observed [Ho11]. Such boxes interfere with the use of the EDO Extension option in the SYN and SYN/ACK segments because extended option space would be misinterpreted as user data if the EDO Extension option were removed, and this cannot be avoided. This is one reason that SYN and SYN/ACK extension requires alternate

mechanisms (see Section 8.7). It is also the reason for the 6-byte EDO Extension variant (see Section 5.3), which can detect such merging or splitting of segments. Further, if such middleboxes become present on a path they could cause similar misinterpretation on segments exchanged in the ESTABLISHED and subsequent states. As a result, this document requires that the EDO Extension option be avoided on the SYN/ACK and that this option needs to be used on all segments once successfully negotiated and encourages use of the 6-byte EDO Extension variant.

Deep-packet inspection systems that inspect TCP segment payloads or attempt to reconstitute the data stream would incorrectly include option data in the reconstituted user data stream, which might interfere with their operation.

>> It can be important to detect misbehavior that could cause EDO space to be misinterpreted as user data. In such cases, EDO SHOULD be used in conjunction with an integrity protection mechanism. This includes the 6-byte EDO Extension variant or stronger mechanisms such as IPsec, TCP-AO, etc. It is useful to note that such protection only helps non-compliant components and enable avoidance (e.g., disabling EDO), but integrity protection alone cannot correct the misinterpretation of EDO space as user data.

This situation is similar to that of ECN and ICMP support in the Internet. In both cases, endpoints have evolved mechanisms for detecting and robustly operating around "black holes". Very similar algorithms are expected to be applicable for EDO.

8. Comparison to Previous Proposals

EDO is the latest in a long line of attempts to increase TCP option space [Al06][Ed08][Ko04][Ra12][Yo11]. The following is a comparison of these approaches to EDO, based partly on a previous summary [Ra12]. This comparison differs from that summary by using a different set of success criteria.

8.1. EDO Criteria

Our criteria for a successful solution are as follows:

- Zero-cost fallback to legacy endpoints.
- Minimal impact on middlebox compatibility.
- No additional side-effects.

Zero-cost fallback requires that upgraded hosts incur no penalty for attempting to use EDO. This disqualifies dual-stack approaches, because the client might have to delay connection establishment to wait for the preferred connection mode to complete. Note that the impact of legacy endpoints that silently reflect unknown options are not considered, as they are already non-compliant with existing TCP requirements [RFC793].

Minimal impact on middlebox compatibility requires that EDO works through simple NAT and NAPT boxes, which modify IP addresses and ports and recompute IPv4 header and TCP segment checksums. Middleboxes that reject unknown options or that process segments in detail without regard for unknown options are not considered; they process segments as if they were an endpoint but do so in ways that are not compliant with existing TCP requirements (e.g., they should have rejected the initial SYN because of its unknown options rather than silently relaying it).

EDO also attempts to avoid creating side-effects, such as might happen if options were split across multiple TCP segments (which could arrive out of order or be lost) or across different TCP connections (which could fail to share fate through firewalls or NAT/NAPTs).

These requirements are similar to those noted in [Ra12], but EDO groups cases of segment modification beyond address and port - such as rewriting, segment drop, sequence number modification, and option stripping - as already in violation of existing TCP requirements regarding unknown options, and so we do not consider their impact on this new option.

8.2. Summary of Approaches

There are three basic ways in which TCP option space extension has been attempted:

1. Use of a TCP option.

2. Redefinition of the existing TCP header fields.

3. Use of option space in multiple TCP segments (split across multiple segments).

A TCP option is the most direct way to extend the option space and is the basis of EDO. This approach cannot extend the option space of the initial SYN.

Redefining existing TCP header fields can be used to either contain additional options or as a pointer indicating alternate ways to interpret the segment payload. All such redefinitions make it difficult to achieve zero-impact backward compatibility, both with legacy endpoints and middleboxes.

Splitting option space across separate segments can create unintended side-effects, such as increased delay to deal with path latency or loss differences.

The following discusses three of the most notable past attempts to extend the TCP option space: Extended Segments, TCPx2, LO/SLO, and LOIC. [Ra12] suggests a few other approaches, including use of TCP option cookies, reuse/overload of other TCP fields (e.g., the URG pointer), or compressing TCP options. None of these is compatible with legacy endpoints or middleboxes.

8.3. Extended Segments

TCP Extended Segments redefined the meaning of currently unused values of the Data Offset (DO) field [Ko04]. TCP defines DO as indicating the length of the TCP header, including options, in 32-bit words. The default TCP header with no options is 5 such words, so the minimum currently valid DO value is 5 (meaning 40 bytes of option space). This document defines interpretations of values 0-4: DO=0 means 48 bytes of option space, DO=1 means 64, DO=2 means 128, DO=3 means 256, and DO=4 means unlimited (e.g., the entire payload is option space). This variant negotiates the use of this capability by using one of these invalid DO values in the initial SYN.

Use of this variant is not backward-compatible with legacy TCP implementations, whether at the desired endpoint or on middleboxes. The variant also defines a way to initiate the feature on the passive side, e.g., using an invalid DO during the SYN/ACK when the initial SYN had a valid DO. This capability allows either side to initiate use of the feature but is also not backward compatible.

8.4. TCPx2

TCPx2 redefines legacy TCP headers by basically doubling all TCP header fields [Al06]. It relies on a new transport protocol number to indicate its use, defeating backward compatibility with all existing TCP capabilities, including firewalls, NATs/NAPTs, and legacy endpoints and applications.

8.5. LO/SLO

The TCP Long Option (LO, [Ed08]) is very similar to EDO, except that presence of LO results in ignoring the existing Data Offset (DO) field and that LO is required to be the first option. EDO considers the need for other fields to be first and declares that the EDO is the last option as indicated by the DO field value. Like LO, EDO is required in every segment once negotiated.

The TCP Long Option draft also specified the SYN Long Option (SLO) [Ed08]. If SLO is used in the initial SYN and successfully negotiated, it is used in each subsequent segment until all of the initial SYN options are transmitted.

LO is backward compatible, as is SLO; in both cases, endpoints not supporting the option would not respond with the option, and in both cases the initial SYN is not itself extended.

SLO does modify the three-way handshake because the connection isn't considered completely established until the first data byte is acknowledged. Legacy TCP can establish a connection even in the absence of data. SLO also changes the semantics of the SYN/ACK; for legacy TCP, this completes the active side connection establishment, where in SLO an additional data ACK is required. A connection whose initial SYN options have been confirmed in the SYN/ACK might still fail upon receipt of additional options sent in later SLO segments. This case - of late negotiation fail - is not addressed in the specification.

8.6. LOIC

TCP Long Options by Invalid Checksum is a dual-stack approach that uses two initial SYNS to initiate all updated connections [Yo11]. One SYN negotiates the new option and the other SYN payload contains only the entire options. The negotiation SYN is compliant with existing procedures, but the option SYN has a deliberately incorrect TCP checksum (decremented by 2). A legacy endpoint would discard the segment with the incorrect checksum and respond to the negotiation SYN without the LO option.

Use of the option SYN and its incorrect checksum both interfere with other legacy components. Segments with incorrect checksums will be silently dropped by most middleboxes, including NATs/NAPTs. Use of two SYNs creates side-effects that can delay connections to upgraded endpoints, notably when the option SYN is lost or the SYNs arrive out of order. Finally, by not allowing other options in the negotiation SYN, all connections to legacy endpoints either use no

options or require a separate connection attempt (either concurrent or subsequent).

8.7. Problems with Extending the Initial SYN

The key difficulty with most previous proposals is the desire to extend the option space in all TCP segments, including the initial SYN, i.e., SYN with no ACK, typically the first segment of a connection, as well as possibly the SYN/ACK. It has proven difficult to extend space within the segment of the initial SYN in the absence of prior negotiation while maintaining current TCP three-way handshake properties, and it may be similarly challenging to extend the SYN/ACK (depending on asymmetric middlebox assumptions).

A new TCP option cannot extend the Data Offset of a single TCP initial SYN segment, and cannot extend a SYN/ACK in a single segment when considering misbehaving middleboxes. All TCP segments, including the initial SYN and SYN/ACK, may include user data in the payload data [RFC793], and this can be useful for some proposed features such as TCP Fast Open [RFC7413]. Legacy endpoints that ignore the new option would process the payload contents as user data and send an ACK. Once ACK'd, this data cannot be removed from the user stream.

The Reserved TCP header bits cannot be redefined easily, even though three of the six total bits have already been redefined (ECE/CWR [RFC3168] and NS [RFC3540]). Legacy endpoints have been known to reflect received values in these fields; this was safely dealt with for ECN but would be difficult here [RFC3168].

TCP initial SYN (SYN and not ACK) segments can use every other TCP header field except the Acknowledgement number, which is not used because the ACK field is not set. In all other segments, all fields except the three remaining Reserved header bits are actively used. The total amount of available header fields, in either case, is insufficient to be useful in extending the option space.

The representation of TCP options can be optimized to minimize the space needed. In such cases, multiple Kind and Length fields are combined, so that a new Kind would indicate a specific combination of options, whose order is fixed and whose length is indicated by one Length field. Most TCP options use fields whose size is much larger than the required Kind and Length components, so the resulting efficiency is typically insufficient for additional options.

The option space of an initial SYN segment might be extended by using multiple initial segments (e.g., multiple SYNs or a SYN and non-SYN) or based on the context of previous or parallel connections. This method may also be needed to extend space in the SYN/ACK in the presence of misbehaving middleboxes. Because of their potential complexity, these approaches are addressed in separate documents [Bo14][Br14][To18].

Option space cannot be extended in outer layer headers, e.g., IPv4 or IPv6. These layers typically try to avoid extensions altogether, to simplify forwarding processing at routers. Introducing new shim layers to accommodate additional option space would interfere with deep-packet inspection mechanisms that are in widespread use.

As a result, EDO does not attempt to extend the space available for options in TCP initial SYNs. It does extend that space in all other segments (including SYN/ACK), which has always been trivially possible once an option is defined.

9. Implementation Issues

TCP segment processing can involve accessing nonlinear data structures, such as chains of buffers. Such chains are often designed so that the maximum default TCP header (60 bytes) fits in the first buffer. Extending the TCP header across multiple buffers may necessitate buffer traversal functions that span boundaries between buffers. Such traversal can also have a significant performance impact, which is additional rationale for using TCP option space - even extended option space - sparingly.

Although EDO can be large enough to consume the entire segment, it is important to leave space for data so that the TCP connection can make forward progress. It would be wise to limit EDO to consuming no more than MSS-4 bytes of the IP segment, preferably even less (e.g., MSS-128 bytes).

When using the ExID variant for testing and experimentation, either TCP option codepoint (253, 254) is valid in sent or received segments.

Implementers need to be careful about the potential for offload support interfering with this option. The EDO data needs to be passed to the protocol stack as part of the option space, not integrated with the user segment, to allow the offload to independently determine user data segment boundaries and combine them correctly with the extended option data. Some legacy hardware receive offload engines may present challenges in this regard, and

may be incompatible with EDO where they incorrectly attempt to process segments with unknown options. Such offload engines are part of the protocol stack and updated accordingly. Issues with incorrect resegmentation by an offload engine can be detected in the same way as middlebox tampering.

10. Security Considerations

It is meaningless to have the Data Offset further exceed the position of the EDO data offset option.

>> When the EDO Extension option is present, the EDO Extension option SHOULD be the last non-null option covered by the TCP Data Offset, because it would be the last option affected by Data Offset.

This also makes it more difficult to use the Data Offset field as a covert channel.

11. IANA Considerations

We request that, upon publication, this option be assigned a TCP Option codepoint by IANA, which the RFC Editor will replace EDO-OPT in this document with codepoint value.

The TCP Experimental ID (ExID) with a 16-bit value of 0x0ED0 (in network standard byte order) has been assigned for use during testing and preliminary experiments.

12. References

12.1. Normative References

[RFC793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

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

12.2. Informative References

[Al06] Allman, M., "TCPx2: Don't Fence Me In", draft-allman-tcpx2-hack-00 (work in progress), May 2006.

[Bo14] Borman, D., "TCP Four-Way Handshake", draft-borman-tcp4way-00 (work in progress), October 2014.

[Br14] Briscoe, B., "Inner Space for TCP Options", draft-briscoe-tcpm-inner-space-01 (work in progress), October 2014.

[Ed08] Eddy, W. and A. Langley, "Extending the Space Available for TCP Options", draft-eddy-tcp-loo-04 (work in progress), July 2008.

[Ho11] Honda, M., Nishida, Y., Raiciu, C., Greenhalgh, A., Handley, M., and H. Tokuda, "Is it still possible to extend TCP", Proc. ACM Sigcomm Internet Measurement Conference (IMC), 2011, pp. 181-194.

[Ko04] Kohler, E., "Extended Option Space for TCP", draft-kohler-tcpm-extopt-00 (work in progress), September 2004.

[Ni15] Nishida, Y., "A-PAWS: Alternative Approach for PAWS", draft-nishida-tcpm-apaws-02 (work in progress), Oct. 2015.

[Ra12] Ramaiah, A., "TCP option space extension", draft-ananth-tcpm-tcpoptext-00 (work in progress), March 2012.

[RFC792] Postel, J., "Internet Control Message Protocol", RFC 792, September 1981.

[RFC1812] Baker, F. (Ed.), "Requirements for IP Version 4 Routers," RFC 1812, June 1995.

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

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, September 2001.

[RFC3234] Carpenter, B. and S. Brim, "Middleboxes: Taxonomy and Issues", RFC 3234, February 2002.

[RFC3540] Spring, N., Wetherall, D., and D. Ely, "Robust Explicit Congestion Notification (ECN) Signaling with Nonces", RFC 3540, June 2003.

[RFC5482] Eggert, L., and F. Gont, "TCP User Timeout Option", RFC 5482, March 2009.

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

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

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

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

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

[To18] Touch, J., T. Faber, "TCP SYN Extended Option Space Using an Out-of-Band Segment", draft-touch-tcpm-tcp-syn-ext-opt (work in progress), Jan. 2018.

[Yo11] Yourtchenko, A., "Introducing TCP Long Options by Invalid Checksum", draft-yourtchenko-tcp-loic-00 (work in progress), April 2011.

13. Acknowledgments

The authors would like to thank the IETF TCPM WG for their feedback, in particular: Oliver Bonaventure, Bob Briscoe, Ted Faber, John Leslie, Pasi Sarolahti, Richard Scheffenegger, and Alexander Zimmerman.

This work is partly supported by USC/ISI's Postel Center.

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

Authors' Addresses

Joe Touch

Manhattan Beach, CA 90266 USA

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

Wesley M. Eddy
MTI Systems
US

Email: wes@mti-systems.com

Network Working Group N. Khademi
Internet-Draft M. Welzl
Updates: 3168 (if approved) University of Oslo
Intended status: Experimental G. Armitage
Expires: October 5, 2016 Swinburne University of Technology
G. Fairhurst
University of Aberdeen
April 03, 2016

TCP Alternative Backoff with ECN (ABE)
draft-khademi-alternativebackoff-ecn-03

Abstract

This memo provides an experimental update to RFC3168. It updates the TCP sender-side reaction to a congestion notification received via Explicit Congestion Notification (ECN). The updated method reduces cwnd by a smaller amount than TCP does in 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 discuss SCTP as well as other transports using ECN.

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 October 5, 2016.

Copyright Notice

Copyright (c) 2016 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction 2
2. Discussion 3
2.1. Why use ECN to vary the degree of backoff? 3
2.2. Choice of ABE multiplier 4
3. NEW: Updating the Sender-side ECN Reaction 5
3.1. RFC 2119 6
3.2. Update to RFC 3168 6
3.3. Status of the Update 7
4. Acknowledgements 7
5. IANA Considerations 7
6. Security Considerations 7
7. References 8
7.1. Normative References 8
7.2. Informative References 8
Authors' Addresses 9

1. Introduction

Explicit Congestion Notification (ECN) is specified in [RFC3168]. It allows a network device that uses Active Queue Management (AQM) to set the congestion experienced, CE, codepoint in the ECN field of the IP packet header, rather than drop ECN-capable packets when incipient congestion is detected. When an ECN-capable transport is used over a path that supports ECN, it provides the opportunity for flows to improve their performance in the presence of incipient congestion [I-D.AQM-ECN-benefits].

[RFC3168] not only specifies the router use of the ECN field, it also specifies a TCP procedure for using ECN. This states that a TCP sender should treat the ECN indication of congestion in the same way as that of a non-ECN-Capable TCP flow experiencing loss, by halving the congestion window "cwnd" and by reducing the slow start threshold "ssthresh". [RFC5681] stipulates that TCP congestion control sets "ssthresh" to max(FlightSize / 2, 2*SMSS) in response to 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 multipler of 0.5 (halving cwnd and sshthresh after packet loss) is not the only available strategy. As defined in [ID.CUBIC], CUBIC multiplies the current cwnd by 0.8 in response to loss (although the Linux implementation of CUBIC has used a multiplier of 0.7 since kernel version 2.6.25 released in 2008). Consequently, CUBIC utilise 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].

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 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.

Results reported in [ABE2015] show significant benefits (improved throughput) when reacting to ECN-Echo by multiplying cwnd and sstthresh with a value in the range [0.7..0.85]. Section 2 describes the rationale for this change. Section 3 specifies a change to the TCP sender backoff behaviour in response to an indication that CE-marks have been received by the receiver.

2. Discussion

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 in reaction to 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."

2.1. Why use ECN to vary the degree of backoff?

The classic rule-of-thumb dictates 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 enough 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 (lmany concurrent TCP connections). However, it interacts badly with lightly-multiplexed cases (few concurrent connections) over high BDP paths. Conventional TCP backoff in such cases leads to gaps in packet transmission and under-utilisation of the path.

In an ideal world, the TCP sender would adapt its backoff strategy to match the effective depth at which a bottleneck begins indicating congestion. In the practical world, [ABE2015] proposes using the existence of ECN CE-marks to infer whether a path's bottleneck is AQM-enabled (shallow queue) or classic DropTail (deep queue), and adjust backoff accordingly. This results in a change to [RFC3168], which recommended that TCP senders respond in the same way following indication of a received ECN CE-mark and a packet loss, making these equivalent signals of congestion. (The idea to change this behaviour pre-dates ABE. [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 deal with RED-based bottlenecks that were not necessarily configured to emulate a shallow queue.)

[RFC7567] states that "deployed AQM algorithms SHOULD support Explicit Congestion Notification (ECN) as well as loss to signal congestion to endpoints" and [I-D.AQM-ECN-benefits] encourages this deployment. Apple recently announced their intention to enable ECN in iOS 9 and OS X 10.11 devices [WWDC2015]. By 2014, server-side ECN negotiation was observed to be provided by the majority of the top million web servers [PAM2015], and only 0.5% of websites incurred additional connection setup latency using RFC3168-compliant ECN-fallback mechanisms.

2.2. Choice of ABE multiplier

ABE decouples a TCP sender's reaction to loss and ECN CE-marks. The description respectively uses beta_{loss} and beta_{ecn} to refer to the multiplicative decrease factors applied in response to packet loss and in response to an indication of a received CN CE-mark 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_*, 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 we want to optimise the reponse of a TCP connection to shallow AQM marking thresholds. 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 (c.f., beta_{loss} = 0.5).

3. NEW: Updating the Sender-side ECN Reaction

This section specifies an experimental update to [RFC3168].

3.1. RFC 2119

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

3.2. Update to RFC 3168

This document specifies an update to the TCP sender reaction that follows when the TCP receiver signals that ECN CE-marked packets have been received.

The first paragraph of Section 6.1.2, "The TCP Sender", in [RFC3168] contains the following text:

"If the sender receives an ECN-Echo (ECE) ACK packet (that is, an ACK packet with the ECN-Echo flag set in the TCP header), then the sender knows that congestion was encountered in the network on the path from the sender to the receiver. The indication of congestion should be treated just as a congestion loss in non-ECN-Capable TCP. That is, the TCP source halves the congestion window "cwnd" and reduces the slow start threshold "ssthresh"."

This memo updates this by replacing it with the following text:

"If the sender receives an ECN-Echo (ECE) ACK packet (that is, an ACK packet with the ECN-Echo flag set in the TCP header), then the sender knows that congestion was encountered in the network on the path from the sender to the receiver. This indication of congestion could be treated in the same way as a congestion loss, however reception of the ECN-Echo flag SHOULD produce a reduction in FlightSize that is less than the reduction had the flow experienced loss. The reduction needs to be sufficient to allow flows sharing a bottleneck to increase their share of the capacity. This reduction MUST be less than 0.85 (at least a 15% reduction).

An ECN-capable network device cannot eliminate the possibility of loss, because a drop may occur due to a traffic burst exceeding the instantaneous available capacity of a network buffer or as a result of the AQM algorithm (overload protection mechanisms, etc [RFC7567]). Whatever the cause of loss, detection of a missing packet needs to trigger the standard loss-based congestion control response. This explicitly does not update this behaviour.

In addition, this document RECOMMENDS that experimental deployments method multiply the FlightSize by 0.8 and reduce the slow start threshold 'ssthresh' in response to reception of a TCP segment that sets the ECN-Echo flag."

3.3. 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.

4. 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, 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).

5. IANA Considerations

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

This memo includes no request to IANA.

6. Security Considerations

The described method is a sender-side only transport change, and does not change the protocol messages exchanged. The security considerations of RFC 3819 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 [ID.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.

7. References

7.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <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>.

7.2. 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.CoDel]
Nichols, K., Jacobson, V., McGregor, V., and J. Iyengar, "The Benefits of using Explicit Congestion Notification (ECN)", Internet-draft, IETF work-in-progress draft-ietf-aqm-codel-02, December 2015.

[I-D.PIE] Pan, R., Natarajan, P., Baker, F., White, G., VerSteeg, B., Prabhu, M., Piglione, C., and V. Subramanian, "PIE: A Lightweight Control Scheme To Address the Bufferbloat Problem", Internet-draft, IETF work-in-progress draft-ietf-aqm-pie-03, November 2015.

[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>.

[ID.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-00, June 2015.

[PAM2015] Trammell, B., Kuhlewind, M., Boppart, D., Learmonth, I., Fairhurst, G., and R. Scheffenegger, "Enabling Internet-wide Deployment of Explicit Congestion Notification", Proceedings of the 2015 Passive and Active Measurement Conference, New York, March 2015, <http://ecn.ethz.ch/ecn-pam15.pdf>.

[WWDC2015]
Lakhera, P. and S. Cheshire, "Your App and Next Generation Networks", Apple Worldwide Developers Conference 2015, San Francisco, USA, June 2015, <https://developer.apple.com/videos/wwdc/2015/?id=719>.

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

TCP Maintenance and Minor Extensions (tcpm)
Internet-Draft
Intended status: Standards Track
Expires: January 1, 2016
A. Zimmermann
R. Scheffenegger
NetApp, Inc.
B. Briscoe

June 30, 2015

The TCP Echo and TCP Echo Reply Options
draft-zimmermann-tcpm-echo-option-00

Abstract

This document specifies the TCP Echo and TCP Echo Reply options. It provides a single field a TCP sender can use to store any type of data that a TCP receiver simply echo unmodified back. In contrast to the original TCP Echo and TCP Echo Reply options defined in RFC 1072 the options specified in this document have slightly different semantics and support a variable option length.

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 1, 2016.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must

include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

1. Introduction

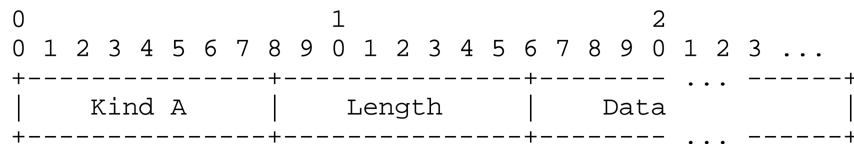
This document specifies the TCP Echo and TCP Echo Reply options. It provides a single field a TCP sender can use to store any type of data that a TCP receiver simply echo unmodified back. In contrast to the original TCP Echo and TCP Echo Reply options defined in RFC 1072 [RFC1072] the options specified in this document have a slightly different semantics and support a variable option length.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119]. These words only have such normative significance when in ALL CAPS, not when in lower case.

3. The TCP Echo and TCP Echo Reply options

The general structure of TCP options is defined in [RFC0793]. The TCP Echo option is organized as indicated in Figure 1.



```
 0                   1                   2
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 ...
+---------------+---------------+-------- ... ------+
|    Kind A     |    Length     |   Data            |
+---------------+---------------+-------- ... ------+
```

Figure 1: The TCP Echo option

The codepoint value of the TCP Echo 'Kind A' is {ToDo: Value TBA}. The value of the 'Length' field in octets can be any value greater than 1 as long as the TCP Echo option completely fits into TCP option space, which may be extended (see [RFC0793], [I-D.ietf-tcpm-tcp-edo], [I-D.briscoe-tcpm-inner-space]). The optional 'Data' field is available for the TCP sender to fill with any amount of any type of data it wishes to be send back by the TCP receiver in a subsequent TCP Echo Reply option (see Figure 2). It is only be constrained in size to an integer number of octets.

The TCP Echo facility is determined in both directions using a single exchange during the 3-way handshake [RFC0793]. A TCP seeking to use TCP Echo facility includes the TCP Echo option in the initial SYN or SYN/ACK. If the TCP receiver of that SYN or SYN/ACK agrees to

support TCP Echo facility, it MUST respond with TCP Echo Reply option (see Figure 2) in its corresponding segment.

Both TCP endpoints MAY use the TCP Echo facility in any segment, but only if the TCP Echo option was received in a segment with the SYN bit set (i.e., SYN and SYN/ACK) or the TCP Echo Reply option was received in response to a sent TCP Echo option. In all cases an endpoint MUST NOT include more than one TCP Echo option per segment.

A TCP sender MAY send an empty TCP Echo option with Length=2 on the SYN, to only indicate that it supports the TCP Echo facility. In that case, the TCP receiver of that SYN MUST response with and empty TCP Echo Reply option with Length=2 accordingly.

The TCP Echo Reply option is organized as indicated in Figure 2.

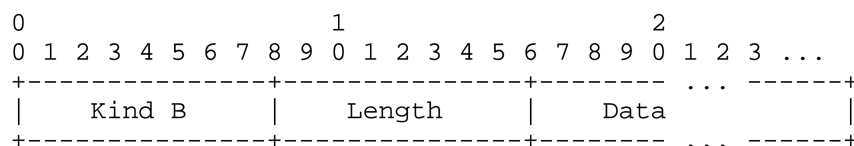


Figure 2: The TCP Echo Reply Option

A TCP receiver that does not implement the TCP Echo facility or decides to not use the TCP Echo facility for this particular connection MUST silently ignore any TCP Echo options it receives for this connection. If the TCP receiver has reflected the TCP Echo option in its SYN/ACK during the 3-way handshake, it MUST reply to any TCP Echo option received during this connection.

Once enabled on a connection, a TCP receiver that receives a TCP Echo option MUST return the same bytes of the Data field in a TCP Echo Reply option. This TCP Echo Reply option MUST returned in the next segment (e.g., an ACK segment) that is sent. If due to the delayed ACK algorithm [RFC1122] more than one TCP Echo option is received before a reply segment is sent, the TCP receiver MUST choose only one of the options to echo, ignoring the others; specifically, it MUST choose the most recently received TCP Echo option to echo back (i.e. Last In, First Out - LIFO).

4. IANA Considerations

This specification requires IANA to allocate a value from the TCP option kind name-space against the name

'Kind A'
'Kind B'

Early implementation before the IANA allocation MUST follow [RFC6994] and use experimental option 254 and respective Experiment ID:

0xEC01 (16 bits) for the TCP Echo option;
0xEC02 (16 bits) for the TCP Echo Reply option;

The Echo option defined in RFC1072 [RFC1072] specifies different semantics, which do not lend themselves for reuse. Specifically, RFC1072 [RFC1072] specifies to select the TCP Echo option data from the newest segment with the oldest sequence number, while herein we specify to return the TCP Echo option of the most recently received segment, regardless of sequence numbers.

{ToDo: Values TBA and register them with IANA} then migrate to the assigned option after allocation.}

5. Security Considerations

An implementation should not rely on this facility for critical TCP mechanisms, before ensuring that the TCP Echo option data field is reflected back properly and unmodified. If the TCP Echo option is considered critical, a TCP mechanism should have means to verify the integrity of the data contained in the TCP Echo Reply option. Additionally, a malicious receiver or network device may infer the utility of the data in a TCP Echo option, and interpret it for its purposes. A designer using the TCP Echo facility needs to consider this, and take appropriate measures to prevent misuse of the data sent.

Since TCP options are not delivered reliably, a TCP Echo or TCP Echo Reply option may be lost or reordered at any time, a TCP mechanisms MUST to deal appropriately with this occurrences.

If multiple TCP mechanisms want to make use of the TCP Echo facility, the implementer should accommodate for that, for example by encoding the multiple inputs accordingly into the data field of the TCP Echo option.

Some middleboxes have been known to remove TCP options unknown to them like those described in this document (see [Honda11]). As the TCP Echo and TCP Echo Reply option use two different option numbers, it is conceivable that only one or the other may get stripped from a segment, in one direction, resulting in an unidirectional usability of the TCP Echo facility.

6. Privacy Considerations

This document describes a new mechanism to tag individual TCP segments. However, the TCP options described do not expose individual user's data. In order to better maintain the confidentiality of data exchanged on the wire, and to address some aspects of security, it is NOT RECOMMENDED to send easily decipherable data in the clear as data in the TCP Echo option.

7. Acknowledgements

Alexander Zimmermann have received funding from the European Union's Horizon 2020 research and innovation program 2014-2018 under grant agreement No. 644866 (SSICLOPS). This document reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

8. References

8.1. Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[RFC1122] Braden, R., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, October 1989.

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

[RFC6994] Touch, J., "Shared Use of Experimental TCP Options", RFC 6994, August 2013.

8.2. Informative References

[Honda11] Honda, M., Nishida, Y., Raiciu, C., Greenhalgh, A., Handley, M., and H. Tokuda, "Is it still possible to extend TCP?", Proc. of ACM Internet Measurement Conference (IMC) '11, November 2011.

[I-D.briscoe-tcpm-inner-space] Briscoe, B., "Inner Space for TCP Options", draft-briscoe-tcpm-inner-space-01 (work in progress), October 2014.

[I-D.ietf-tcpm-tcp-edo] Touch, J. and W. Eddy, "TCP Extended Data Offset Option", draft-ietf-tcpm-tcp-edo-01 (work in progress), October 2014.

[RFC1072] Jacobson, V. and R. Braden, "TCP extensions for long-delay paths", RFC 1072, October 1988.

Authors' Addresses

Alexander Zimmermann
NetApp, Inc.
Sonnenallee 1
Kirchheim 85551
Germany

Phone: +49 89 900594712
Email: alexander.zimmermann@netapp.com

Richard Scheffenegger
NetApp, Inc.
Am Euro Platz 2
Vienna 1120
Austria

Email: rs@netapp.com

Bob Briscoe

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

TCP Maintenance and Minor Extensions (tcpm)
Internet-Draft
Intended status: Standards Track
Expires: January 21, 2016

A. Zimmermann
R. Scheffenegger
NetApp, Inc.
July 20, 2015

Using the TCP Echo Option for Spurious Retransmission Detection
draft-zimmermann-tcpm-spurious-rxmit-00

Abstract

The Spurious Retransmission Detection (SRD) algorithm allows a TCP sender to always detect if it has entered loss recovery unnecessarily. It requires that both the TCP Echo option defined in [I-D.zimmermann-tcpm-echo-option], and the SACK option [RFC2018] be enabled for a connection. The SRD algorithm makes use of the fact that the TCP Echo option, used in conjunction with the SACK feedback, can be used to completely eliminate the retransmission ambiguity in TCP. Based on the reflected data contained in the first acceptable ACK that arrives during loss recovery, it decides whether loss recovery was entered unnecessarily. The SRD mechanism further enables improvements in loss recovery. This includes a TCP enhancement to detect and quickly resend lost retransmissions.

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 21, 2016.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents

(http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction 2
2. Terminology 3
3. The Spurious Retransmission Detection Algorithm 3
  3.1. Motivation 4
  3.2. Basic Idea 5
  3.3. The Algorithm 6
4. Examples 7
5. IANA Considerations 12
6. Security Considerations 12
7. Acknowledgements 13
8. References 13
  8.1. Normative References 13
  8.2. Informative References 13
Authors' Addresses 14

1. Introduction

Using only the sequence number, a TCP sender is not able to distinguish whether the first ACK, acknowledging new data, that arrives after a retransmit, was sent in response to the original transmit or the retransmission. This effect is known as the retransmission ambiguity problem [Zh86], [KP87]. Spurious retransmissions, where a segment is sent multiple times, can be caused by packet reordering, packet duplication, or a sudden delay increase in the data or the ACK path. All these cases are preceded by either a fast retransmit or a timeout-based retransmit.

The Eifel Detection Algorithm [RFC3522] aims to address these occurrences, but falls short to completely solve the ambiguity problem due to limitations in how the TCP Timestamps option is processed by the receiver.

The TCP Timestamps option already provides a means of marking retransmitted segments differently. However, the method used by a TCP receiver when a Timestamp option is reflected precludes the use of this option in most cases. The notable exception is the recovery of lost segments, when none of the retransmissions is lost or reordered in turn. Similarly, spurious retransmissions can also only

be detected and recovered from, when all of the retransmitted packets are delivered in-order and without leaving any gaps in the receive-buffer. Elsewise, the Timestamp option does not allow a solid discrimination between original or retransmitted segments, that triggered subsequent duplicate ACKs.

The semantics of the TCP Echo option, and their treatment by a receiver are different from those of the TCP Timestamps option. That allows a complete solution to disambiguate between all retransmissions, including multiple retransmissions of the same segment, packet duplication, and reordering events.

Enhancements in the area of TCP loss recovery and spurious retransmission detection are allowed by using synergistic signaling between the TCP Echo option and the selective acknowledgment (SACK) option. This allows to completely address any retransmission ambiguity.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119]. These words only have such normative significance when in ALL CAPS, not when in lower case.

Acceptable ACK: is an ACK that acknowledges previously unacknowledged data. See [RFC0793].

Forward Acknowledgement (FACK): is the the highest sequence number known to have reached the receiver, plus one, using SACK information. See [MM96].

Lost Retransmission Detection (LRD): is a mechanism to timely detect lost retransmissions during loss recovery, and quickly send the lost segment anew instead of waiting for a retransmission timeout. A simple and limited variant, that is not formally specified, is currently in use by the Linux TCP stack.

Recover: When in fast recovery, this variable records the send sequence number that must be acknowledged before the fast recovery procedure is declared to be over. See [RFC6582].

3. The Spurious Retransmission Detection Algorithm

3.1. Motivation

In order to detect spurious retransmissions, the sender requires information to uniquely identify each retransmission of every segment sent. TCP Eifel [RFC3522] uses additional information from the TCP Timestamps option [RFC7323] for this purpose. This can remove some ambiguity, but only under limited circumstances - it only works in the absence of additional impediments like ACK reordering or multiple loss.

However, the semantics used by the receiver when reflecting back a received timestamp is such that this approach only works for the first retransmission in a window, every subsequent retransmission cannot be disambiguated from a received original transmission using timestamps in most cases.

When a segment is retransmitted without the timestamp clock increasing, Eifel detection also has no signal to differentiate if a spurious retransmission had occurred. This is of particular concern at high data rates and when the RTT is low.

Retransmission ambiguity detection during loss recovery (as opposed to the first retransmission in a window) allows an additional level of loss recovery control without reverting to timer-based methods. As with the deployment of SACK, separating "what" to send from "when" to send it, is driven one step further. In particular, less conservative loss recovery schemes, which do not trade the principle of packet conservation against timeliness, require a reliable way of prompt and best possible feedback from the receiver about any delivered segment and the ordering in which they got delivered.

SACK signaling [RFC2018] goes quite a long way, but does not suffice in all circumstances, e.g. when retransmissions are lost. Further, DSACK [RFC2883] does indicate if spurious retransmissions occured, but that signal is delayed by one RTT [RFC3708]. However, loss recovery is likely to have ended at that time. Furthermore, the DSACK option by itself will not yield the information, if the late arrived segment was the original or retransmitted segment.

Using the facility provided by the TCP Echo option a TCP sender is able to differentiate between original and retransmitted segments, even within the same TCP Timestamps options clock tick (i.e. when RTT is shorter than the TCP timestamp clock interval). In addition, as the TCP Echo option is reflected back with the most recently observed value by the receiver, all instances where Eifel detection [RFC3522] is not able to detect reliably can be addressed. Furthermore, as the sender is immediately notified which segment triggered the ACK, no delay is induced when deducting if a retransmission was spurious.

3.2. Basic Idea

Using the TCP Echo option, which has different semantics from the TCP Timestamps option, it is possible to uniquely identify and disambiguate each segment, including every retransmission. However, the value carried with the TCP Echo option does not need to be unique by itself (e.g. every segment having a different TCP Echo option value), as other information contained in the TCP Header and TCP options, namely the acknowledgment number and the SACK blocks, differentiate already between segments in the TCP stream space. Thus, it is only necessary to differentiate between segments (of the same size) covering the same sequence space.

One simple approach would be to have a per-segment counter, which is set to zero for each new transmission, and incremented whenever that same segment is retransmitted anew. However, this approach would require per-segment state in the sender. To reduce the complexity in the sender, and not require per-segment state, a simpler approach is to use a single global counter, that is increased whenever a segment has to be resent. In ECN environments, an increase of the retransmission counter is expected to typically coincide with CWR-marked segments.

Apart from simplifying the design, this also yields additional benefits when the reorder delay is larger than one RTT, and when Acknowledgments are lost or reordered. Note that the wire representation of this counter SHOULD NOT be as simplistic as described here (see Section 6).

The retransmission counter has to be large enough to cater for all expected RTOs before a TCP sender gives up and terminates a connection (see [RFC1122], section 4.2.3.5, variable R2), plus all the fast retransmissions of that segment that may have happened before triggering the chain of exponential back-off RTOs. In general, a single octet is enough to convey the retransmission counter.

The sender has to transmit every segment with a TCP Echo option. Sending the Echo option only with retransmission has the issue of adding option space, thereby potentially requiring the sender to segment the TCP payload differently (and sending an additional segment) than the original segment. A sender SHOULD therefore add the echo option to every sent segment to simplify the implementation. Sending the TCP Echo option with every segment has the added benefit to make the mechanism tolerate ACK losses.

3.3. The Algorithm

Spurious Retransmission Detection (SRD) utilizes the TCP Echo option [I-D.zimmermann-tcpm-echo-option], which is used with at least one octet of payload. If another algorithm deployed on the sender also uses the TCP Echo option on a TCP connection, it is up to the implementer to combine the necessary signaling of these mechanisms to fit into a single TCP Echo option (e.g. by mapping the Echo option codepoints into a translation table, or extending the length of the TCP Echo option and matching parts of the data to the different mechanisms).

The TCP sender maintains a single, connection-global counter. This retransmission counter MUST be increased by one whenever the sender enters loss recovery, experiences a Retransmission Timeout (RTO), or re-sends a previously already retransmitted segment once more. Care must be taken to limit a malicious receivers ability make genuine retransmissions appear as spurious retransmissions to the sender (see Section 6), when encoding the internal counter value to the wire representation.

Every transmitted segment carries a TCP Echo option, where the data reflects the current value of the sender’s retransmission counter. When the sender receives an ACK, the TCP Echo option data is extracted and checked against the current value of the retransmission counter, together with a check if the ACK is acceptable. Note that information from not acceptable ACKs MUST be evaluated too.

After a retransmission has been sent, either due to a Fast Retransmission or an RTO, the first acceptable ACK is checked. If the received retransmission counter is equal to the current counter value maintained by the sender, a valid retransmission was sent. If the received value is less than the current retransmission counter, a spurious retransmission was sent, and if no valid retransmissions are detected until the end of the loss recovery phase, the TCP sender MAY restore the congestion control state to the state prior to entering loss recovery. Even if some of the retransmissions of this loss recovery phase may have been spurious, the TCP sender MUST NOT react by restoring the congestion control state to the state before entering loss recovery, if any of the retransmissions are deduced to be valid.

A TCP sender MAY retain the congestion control state for up to two RTTs since entering the loss recovery state. {TODO: Not after exiting loss recovery?} If all retransmissions that were performed in this period are later found to have been spurious - either by evaluating the retransmission counter values of received unacceptable (first duplicate) ACKs, or a DSACK [RFC3708] indication - the TCP sender MAY

revert to the stored congestion control state, e.g. by following the Eifel Response algorithm [RFC4015].

4. Examples

This section shows a few examples, from simple to increasingly complex. Some of these scenarios are addressed by exising mechanisms like Eifel, and DSACK; in particular, corner cases that are not adressed with existing mechanisms are demonstrated.

In the following examples, each set of three lines starting with "ack#", "sack:", and "sent:" represent one RTT. It is assumed that the sender has sent segments 1 to 8 in the prior RTT, and for readability, the numbers show represent full segments rather than sequence numbers.

The two lines following ("ack#" and "sack:") indicate what ACK is being triggered on the receiver. The ACK number is the sequence number of the next expected segment, followed by a dot and the value of the received TCP Echo option value - again for simpilicty, the internal representation of the global retransmission counter value (initially set to zero) is shown, not the wire representation.

In the line "sack:" the relevant SACK blocks are depicted, again with a single number representative of an entire segment. When these ACKs are seen by the sender, it will start sending the segment depicted in the line "sent:", again together with the retransmission counter value.

Further assumptions in these examples are that the sender is using proportional rate reduction [RFC6937], limited transmit [RFC3042], and selective acknowledgments (SACK) [RFC2018] and [RFC2883], is not application limited when sending data and has a congestion window of 9 segments.

1. Fast Retransmission

```
ack#   X 1.0  1.0  1.0  1.0  1.0  1.0  1.0
sack:    2    2-3  2-4  2-5  2-6  2-7  2-8
sent:    9.0  10.0 1.1       11.1      12.1

ack#     1.0  1.0  11.1      12.1      13.1
sack:    2-9  2-10
```

detected as valid retransmission, as for the first acceptable ACK (11.1) after the retransmission the Echo Tag is equal to the retransmission counter.

2.   Multiple loss

```
ack#   X 1.0    1.0      1.0  1.0  1.0  1.0    X
sack:    2      2-3      2-4  2-5  2-6  2-7
sent:    9.0    10.0     1.1       11.1

ack#     1.0    1.0      8.1       8.1
sack:    2-7,9  2-7,9-10 9-10      9-11
sent:    12.1            8.1       ...
```

SRD detectes this as valid retransmission, as for the first acceptable ACK (8.1) and every other retransmission after the first retransmission the Echo Tag is equal to the retransmission counter. Retransmission counter is not increased when sending (8.1) as loss recovery was not yet exited at the time of sending that retransmission.

3.   Retransmission Timeout (RTO)

```
ack#  X   X    X    X    X    X    X    X
sack:
sent:    ----- RTO -->

ack#
sack:
sent:    ----- RTO --> 1.1

ack#                   1.1
sack:
```

detected as valid retransmission, as the first acceptable ACK (1.1) after the retransmission contains the Echo Tag of the retransmission.

4.   Retransmission loss

```
ack#   X 1.0  1.0  1.0  1.0  1.0  1.0  1.0
sack:    2    2-3  2-4  2-5  2-6  2-7  2-8
sent:    9.0  10.0 1.1       11.1      12.1
                   X
ack#     1.0  1.0            1.1       1.1
sack:    2-9  2-10           2-11      2-12
```

no acceptable ack, but a jump on the counter tag to the current counter. (see {TODO: LRD document}), also FACK is larger than Recovery Point (The condition of FACK > RP will trigger linux LRD).

Note: without LRD, the lost retransmission will NOT be retried before an RTO. Can not be detected by Eifel due to TCP Timestamps semantics.

5. Multiple loss, first retransmission lost

```
ack#   X X    1.0  1.0  1.0  1.0  1.0  1.0
sack:         3    3-4  3-5  3-6  3-7  3-8
sent:         9.0  1.1       2.1       10.1
                   X
ack#          1.0            1.1       1.1
sack:         3-9            2-9       2-10
sent:         11.1           1.2       12.2
```

no acceptable ack, but a jump on the counter tag to the current counter. see {TODO: LRD document}. Linux LRD would delay the sending of 1.2 until after FACK passes RP (in this example, the last two sent segments was be swapped). Not detectable by Eifel.

6. RTT > Reordering delay > DupThresh

```
                             r
ack#   R 1.0  1.0  1.0  1.0  6.0  7.0  8.0
sack:    2    2-3  2-4  2-5
sent:    8.0  9.0  1.1       10.1 11.1 12.1

ack#     9.0  10.0 10.1      11.1 12.1 13.1
sack:               1
```

detected as spurious retransmission, as the first acceptable ACK (6.0) after the retransmission is received with the Echo Tag unequal the current retransmission counter; DSACK detects this 1 RTT later; Eifel detects this at the same time using timestamps

7. Reordering delay > RTT

```
ack#   R 1.0  1.0  1.0  1.0  1.0  1.0  1.0
sack:    2    2-3  2-4  2-5  2-6  2-7  2-8
sent:    9.0  10.0 1.1       11.1      12.1
                                  r
ack#     1.0  1.0  11.1      12.1 12.0 13.1
sack:    2-9  2-10                1
```

detected as valid retransmission, as the first acceptable ACK (11.1) after the retransmission contains the Echo Tag of the retransmission.

Note that at (12.0), with the retransmission counter always counting up, this detection becomes possible, by seeing 2nd ACK with lower retransmission counter (SRD) one RTT later: DSACK and SRD both detect at the same time

8.   Packet duplication

SACK is mandatory for SRD, and SACK detects this as duplication event, with no further action

9.   Reordering and loss

```
                             r
ack#   R X    1.0  1.0  1.0  2.0  2.0  2.0
sack:         3    3-4  3-5  3-5  3-6  3-7
sent:         8.0  9.0  1.1       2.1

ack#:         2.0  2.0  2.1       10.1
sack          3-8  3-9  1,3-9
```

detected as spurious retransmission, as the first acceptable ACK (2.0) after the retransmission is received with the Echo Tag unequal the current retransmission counter; no undo at that point, since still in recovery. DSACK detects this 1 RTT later; Eifel detects this at the same time using timestamps.
Detected as valid retransmission, as for the second acceptable ACK (10.1) after the retransmission the Echo Tag is equal to the retransmission counter, prior to leaving loss recovery

10. Loss and reordering (reordered retransmission)

```
ack#   X 1.0  1.0  1.0  1.0  1.0  1.0  1.0
sack:    2    2-3  2-4  2-5  2-6  2-7  2-8
sent:    9.0  10.0 1.1       11.1      12.1
                   R
                                  r
ack#     1.0  1.0            1.1  12.1 13.1
sack:    2-9  2-10           2-11
sent:         13.1           1.2  14.2 15.2

ack#          14.1           14.2 15.2 16.2
sack:                        1

                      reordered retransmission
```

LRD triggered (no acceptable ack, when retransmission count increases - {TODO: LRD document}), also FACK > Recovery Point (Linux LRD) Detected as spurious retransmission, as the first acceptable ACK (12.1) after the 2nd retransmission is received with the Echo Tag unequal the current retransmission counter; undo at that point, since recovery is exited at the same time. DSACK detects this 1 RTT later; Eifel detects this at the same time using timestamps.

11. ACK reordering after loss

```
ack#   X 1.0  1.0  1.0  1.0  1.0  1.0  1.0
sack:    2    2-3  2-4  2-5  2-6  2-7  2-8
sent:    9.0  10.0 1.1       11.1      12.1
                   R
                                  r
ack#     1.0  1.0            1.1  11.1 13.1
sack:    2-9  2-10           2-11
sent:         13.1           1.2  14.2 15.2
```

valid retransmission, as first acceptable ack (11.1) after retransmission has same retransmission counter as the current value. Reordered ACK has still same (not lower!) retransmission counter.

12. ACK reordering after reordering

```
                                  rR
ack#   R 1.0  1.0  1.0  1.0  7.0  6.0  8.0
sack:    2    2-3  2-4  2-5
sent:    8.0  9.0  1.1       10.1      11.1

ack#     9.0  10.0 10.1      11.1      12.1
sack:              1
```

detected as spurious retransmission, as the first acceptable ACK (7.0) after the retransmission is received with the Echo Tag unequal the current retransmission counter; DSACK detects this 1 RTT later; Eifel detects this at the same time using timestamps

13. ACK loss after reordering

```
                              r
ack#   R 1.0  1.0  1.0  1.0  (6.0) 7.0  8.0
sack:    2    2-3  2-4  2-5
sent:    8.0  9.0  1.1             10.1 11.1

ack#     9.0  10.0 10.1            11.1 12.1
sack:              1
```

detected as spurious retransmission, as the first acceptable ACK (7.0) after the retransmission is received with the Echo Tag unequal the current retransmission counter; DSACK detects this 1 RTT later; Eifel detects this at the same time using timestamps
Note that retransmission counter only increasing helps this case to work both with reordering (spurious retransmission) and retransmission ACK loss - the relevant information is conveyed for about 1RTT thus single ACK loss does not impact the detection.

14. TODO: delay ACK

Todo: Example necessary?

5. IANA Considerations

This document contains no requests to IANA, as only a new combined use of TCP options is described.

6. Security Considerations

This document describes a new use for the TCP Echo option. Transporting the retransmission counter in the clear may pose a security problem when the TCP sender uses SRD to restore the TCP state. A malicious receiver could game the sender to always restore the congestion control state to the one preceding the lost recovery episode, thereby making the sender not back off its transmission rate.

As the sender can put any data into the TCP Echo option, the transmission counter value can be masked in various ways. A TCP sender can map the same counter value to multiple TCP Echo option data values, and track which of these data values would be expected for a given acknowledgement. Alternatively, the TCP Echo option data could be a (secure) hash of the sequence number of the sent segment, a random, per-connection secret, and the retransmission counter. The TCP Echo data would look rather as random sequence of octets in both cases, making it very hard for a malicious receiver to obtain an unfair share of bandwidth by masking genuine retransmissions as spurious.

7. Acknowledgements

The authors like to thank Bob Briscoe and Brian Trammel for their invaluable input.

Alexander Zimmermann the European Union's Horizon 2020 research and innovation program 2014-2018 under grant agreement No. 644866 (SSICLOPS). This document reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

8. References

8.1. Normative References

[I-D.zimmermann-tcpm-echo-option]
Zimmermann, A., Scheffenegger, R., and B. Briscoe, "The TCP Echo and TCP Echo Reply Options", draft-zimmermann-tcpm-echo-option-00 (work in progress), June 2015.

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

8.2. Informative References

[KP87] Karn, P. and C. Partridge, "Estimating Round-Trip Times in Reliable Transport Protocols", Proc. SIGCOMM '87, August 1987.

[MM96] Mathis, M. and J. Mahdavi, "Forward Acknowledgement: Refining TCP Congestion Control", ACM SIGCOMM 1996 Proceedings, in ACM Computer Communication Review 26 (4), pp. 281-292, October 1996.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[RFC1122] Braden, R., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, October 1989.

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

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

[RFC3042] Allman, M., Balakrishnan, H., and S. Floyd, "Enhancing TCP's Loss Recovery Using Limited Transmit", RFC 3042, January 2001.

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

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

[RFC4015] Ludwig, R. and A. Gurtov, "The Eifel Response Algorithm for TCP", RFC 4015, February 2005.

[RFC6582] Henderson, T., Floyd, S., Gurtov, A., and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 6582, April 2012.

[RFC6937] Mathis, M., Dukkipati, N., and Y. Cheng, "Proportional Rate Reduction for TCP", RFC 6937, May 2013.

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

[Zh86] Zhang, L., "Why TCP timers don't work well", Proc. SIGCOMM '86, Sep 1986.

Authors' Addresses

Alexander Zimmermann
NetApp, Inc.
Sonnenallee 1
Kirchheim 85551
Germany

Phone: +49 89 900594712
Email: alexander.zimmermann@netapp.com

Richard Scheffenegger
NetApp, Inc.
Am Euro Platz 2
Vienna 1120
Austria

Email: rs@netapp.com