TCP Maintenance and Minor Extensions (tcpm)
Internet-Draft
Intended status: Standards Track
Expires: April 28, 2015

B. Briscoe
BT

October 25, 2014

The Echo Cookie TCP Option
draft-briscoe-tcpm-echo-cookie-00

Abstract

This document specifies a TCP Option called EchoCookie. It provides a single field that a TCP server can use to store opaque cookie data 'in flight' rather than in memory. As new TCP options are defined, they can require that implementations support the EchoCookie option. Then if a server's SYN queue is under pressure from a SYN flooding attack, it can ask clients to echo its connection state in their acknowledgement. This facility is similar to the classic SYN Cookie, but it provides enough space for connection state associated with TCP options. In contrast, the classic location for a SYN Cookie only provides enough space for a degraded encoding of the Maximum Segment Size (MSS) TCP option and no others.

Status of This Memo

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

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

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

This Internet-Draft will expire on April 28, 2015.

Table of Contents

1. Introduction

In order to initiate a connection, a TCP client sends a SYN segment to a TCP server. The server normally allocates memory to hold the required connection state then responds with a SYN/ACK segment to the address the client claims to be sending from. If a TCP server is under SYN flood attack, it can resort to including a SYN Cookie in the SYN/ACK [RFC4987] and not holding any connection state until the client follows through with an echo of the SYN Cookie. Therefore, a SYN Cookie effectively allows a TCP server to store its connection state 'in flight' for a round. Then while it is testing which client addresses correctly complete the handshake, it can protect its memory from exhaustion.

The limited size of a SYN Cookie is a known limitation. SYN Cookies are not standardised (and don't need to be), but typically the server encodes its SYN Cookie into the 16 bits of the Initial Sequence Number (ISN) [RFC0793] and the 9 least significant bits of the timestamp option [RFC7323] (if supported by the client). These fields are only large enough to hold a few common TCP options, such as a degraded record of the client's maximum segment size (MSS), the window scale option and SACK-ok. Therefore, SYN Cookies only protect a rudimentary TCP connection service--they do not protect all the facilities provided by TCP options during an attack.

These 41 bits are the only space available for SYN cookies. A server can only exploit fields that it can set to any value it chooses and that are naturally echoed by all (or at least most) TCP clients. Ideally, the server would be able to place a cookie of any reasonable size in a new generic EchoCookie TCP option on the SYN/ACK and the client would be required to echo it back in the following ACK. However, that would be of little use until most clients supported it.

A simple solution to this problem is to require that EchoCookie support must be implemented with any TCP options defined from now on. A new capability to extend the TCP option space on SYN/ACK segments, e.g. [I-D.touch-tcpm-tcp-syn-ext-opt] or [I-D.briscoe-tcpm-inner-space], could also require that the EchoCookie mechanism must be implemented with it.

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]. These words only have such normative significance when in ALL CAPS, not when in lower case.

2. Echo Cookie TCP Option

If a TCP server's SYN queue is under pressure from a SYN flood attack, it MAY send an EchoCookie TCP option on the SYN/ACK, instead of consuming memory to hold connection state.

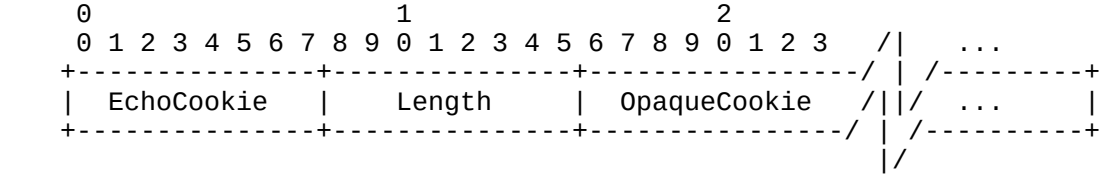


Figure 1: The EchoCookie TCP Option

The general structure of TCP options is defined in [RFC0793]. The EchoCookie TCP option is defined in Figure 1. The Option Kind is EchoCookie with value {ToDo: Value TBA}. The Length in octets can be any value greater than 1.

The OpaqueCookie field is available for the sender to fill with any amount of any type of data it wishes to store in the cookie, only constrained in size to an integer number of octets.

When a TCP receiver acknowledges a segment carrying an EchoCookie option, it MUST return an EchoCookie TCP option carrying an identical OpaqueCookie.

The mechanism a server uses to determine whether the echoed contents of the cookie are the same as the contents it sent are implementation dependent and do not need to be standardised.

The EchoCookie option with length greater than 2 is only defined on a SYN/ACK or on the ACK in response.

A client MAY send an empty EchoCookie TCP option with Length=2 on the SYN, to indicate that it supports the EchoCookie facility. This will not be necessary if support is implied by some other means (e.g. use of the Inner Space protocol [I-D.briscoe-tcpm-inner-space] implies support for EchoCookie).

If there is any TCP Payload in the SYN, it will never be necessary to include this data in a subsequent Echo Cookie. Not acknowledging the data would be sufficient to get the client to retransmit it.

If the client sends a valid TCP Fast Open (TFO) cookie [I-D.ietf-tcpm-fastopen] on the SYN of a resumed connection, there will be no need to defer establishing the connection by responding with an EchoCookie, because the client source address is already known to the server.

3. IANA Considerations

This specification requires IANA to allocate a value from the TCP Option Kind name-space against the name:

"EchoCookie"

Early implementation before the IANA allocation MUST follow [RFC6994] and use experimental option 254 and respective Experiment ID:

0xEEEE (16 bits);

{ToDo: Instead it might be prudent/possible for initial experiments to reuse Option Kinds 6 and/or 7 defined by RFC 1072 (Oct 1988) for a 4-octet Echo and Echo Reply facility that was superceded by the combined Echo and Reply facility in the Timestamp option of RFC1323 (May 1992) and formally obsoleted by RFC6247 (May 2011). Then if the experiments find that no legacy implementations recognise these options it can re-use them to avoid consuming new Option Kind values.}

{ToDo: Values TBA and register them with IANA} then migrate to the assigned option after allocation.}

4. Security Considerations

If the cookie holds state that was negotiated over a secure connection, it MUST be echoed with the same or a stronger level of security.

A SYN/ACK carrying an EchoCookie request MUST NOT exceed the size of the TCP SYN that preceded it. This ensures that the EchoCookie defence cannot amplify an attack by reflection.

A server may record a random selection of the clients to which it responds with an EchoCookie option. Then it can detect if a spoof client is mounting a reflection attack, by repeatedly asking the server to send a SYN/ACK to the same victim client that rarely or never responds. In such a case the server SHOULD limit the frequency at which it responds to such a client.

{ToDo: More?}

5. Acknowledgements

Bob Briscoe’s contribution is part-funded by the European Community under its Seventh Framework Programme through the Trilogy 2 project (ICT-317756). The views expressed here are solely those of the author.

6. References

6.1. Normative References

[I-D.ietf-tcpm-fastopen] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", draft-ietf-tcpm-fastopen-10 (work in progress), September 2014.

[RFC0793] 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.

[RFC6994] Touch, J., "Shared Use of Experimental TCP Options", RFC 6994, August 2013.

6.2. Informative References

[I-D.briscoe-tcpm-inner-space]
Briscoe, B., "Inner Space for TCP Options", draft-briscoe-tcpm-inner-space-00 (work in progress), October 2014.

[I-D.touch-tcpm-tcp-syn-ext-opt]
Touch, J. and T. Faber, "TCP SYN Extended Option Space Using an Out-of-Band Segment", draft-touch-tcpm-tcp-syn-ext-opt-01 (work in progress), September 2014.

[RFC4987] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", RFC 4987, August 2007.

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

Appendix A. Protocol Design Issues (to be Deleted before Publication)

This appendix is informative, not normative. It records outstanding issues with the protocol design that will need to be resolved before publication.

Why limit to SYN/ACK? {ToDo: Consider whether it is OK to generalise EchoCookie with Length > 2 to any segment from client or server (except the SYN, which would create a vulnerability to reflection attacks), especially the FIN, FIN/ACK etc.. It may even be possible to generalise this to cover TFO.}

Author's Address

Bob Briscoe
BT
B54/77, Adastral Park
Martlesham Heath
Ipswich IP5 3RE
UK

Phone: +44 1473 645196
Email: bob.briscoe@bt.com
URI: http://bobbriscoe.net/

TCP Maintenance and Minor Extensions (tcpm)
Internet-Draft
Updates: 793 (if approved)
Intended status: Experimental
Expires: April 30, 2015

B. Briscoe
BT

October 27, 2014

Inner Space for TCP Options
draft-briscoe-tcpm-inner-space-01

Abstract

This document describes an experimental method to extend the limited space for control options in every segment of a TCP connection. It can use a dual handshake so that, from the very first SYN segment, extra option space can immediately start to be used optimistically. At the same time a dual handshake prevents a legacy server from getting confused and sending the control options to the application as user-data. The dual handshake is only one strategy - a single handshake will usually suffice once deployment has got started. The protocol is designed to traverse most known middleboxes including connection splitters, because it sits wholly within the TCP Data. It also provides reliable ordered delivery for control options. Therefore, it should allow new TCP options to be introduced i) with minimal middlebox traversal problems; ii) with incremental deployment from legacy servers; iii) without an extra round of handshaking delay iv) without having to provide its own loss recovery and ordering mechanism and v) without arbitrary limits on available space.

Table of Contents

1. Introduction

TCP has become hard to extend, partly because the option space was limited to 40B when TCP was first defined [RFC0793] and partly because many middleboxes only forward TCP headers that conform to the stereotype they expect.

This specification ensures new TCP capabilities can traverse most middleboxes by tunnelling TCP options within the TCP Data as 'Inner Options' (Figure 1). Then the TCP receiver can reconstruct the Inner Options sent by the sender, even if the middlebox resegments the data stream and even if it strips 'Outer' options from the TCP header that it does not recognise. The two words 'Inner Space' are appropriate as a name for the scheme; 'Inner' because it encapsulates options within the TCP Data and 'Space' because the space within the TCP Data is virtually unlimited--constrained only by the maximum segment size.

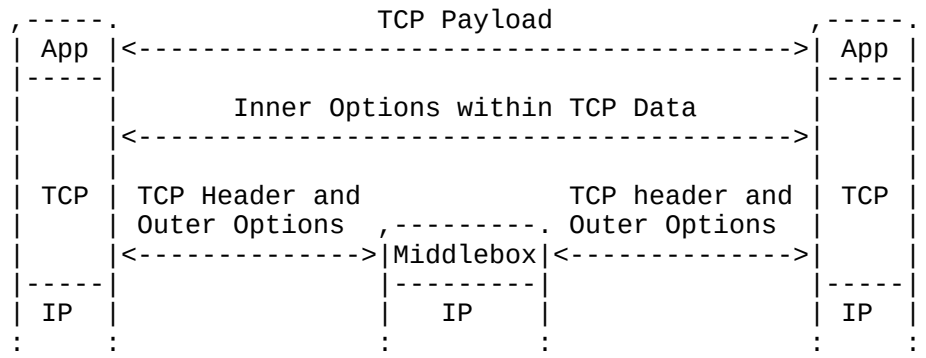


Figure 1: Encapsulation Approach

TCP options fall into three main categories:

a. Those that have to remain as Outer Options--typically those concerned with transmission of each TCP segment, e.g. Timestamps and Selective ACKnowledgements (SACK);

b. Those that are best as Inner Options--typically those concerned with transmission of the data as a stream, e.g. the TCP Authentication Option [RFC5925] or tcpcrypt [I-D.bittau-tcpinc];

c. Those that can be either Inner or Outer Options--typically those used at the start of a connection which is also inherently the start of the first segment so segmentation is not a concern.

Pressure of space is most acute in the initial segments of each half-connection, i.e. the SYN and SYN/ACK, and particularly the SYN. Even though Inner Space is not suitable for category (a) options, moving all of categories (b) and (c) into Inner Space frees up plenty of outer space in the header for category (a).

The following list of options that might be required on a SYN illustrates how acute the problem is:

- 4B: Maximum Segment Size (MSS) [RFC0793];
- 2B: SACK-ok [RFC2018];
- 3B: Window Scale [RFC7323];
- 10B: Timestamp [RFC7323];
- 12B: Multipath TCP [RFC6824];

- 6-18B: TCP Fast Open on a resumed connection [I-D.ietf-tcpm-fastopen];

- 16B: TCP-AO [RFC5925];

There is probably potential for compressing together multiple options in order to mitigate the option space problem. However, the option space problem has to be faced, because complex special placement is already being contemplated for options that can be larger than 40B on their own (e.g. the key agreement options of tcpcrypt [I-D.bittau-tcpinc]).

Given the Inner Space protocol places control options within TCP Data, it is critical that a legacy TCP receiver is never confused into passing this mix to an application as if it were pure data. Naively, both ends could handshake to check they understand the protocol, but this would introduce a round of delay and it would not solve the shortage of space in a SYN. Instead, the client uses dual handshakes; one suitable for an upgraded server, and the other for an ordinary server. Then, if the client discovers that the server does not understand the new protocol, it can abort the upgraded handshake before the server passes corrupt data to the application. Otherwise, if the server does understand the new protocol, the client can abort the ordinary handshake. Either way, it has added zero extra delay. Interworking of the dual handshake with TCP Fast Open [I-D.ietf-tcpm-fastopen] is carefully defined so that either server can pass data to the application as soon as the initial SYN arrives.

When control options are placed within the TCP Data they inherently get delivered reliably and in order. Although this was not originally recognised as part of the design brief, it offers the significant benefit of simplifying the design of new TCP options. Reliable ordered delivery no longer has to be individually crafted into the design of each new TCP option.

Solving the five problems of i) option-space exhaustion; ii) middlebox traversal; iii) legacy server confusion; iv) reliable ordered control message delivery; and v) handshake latency; does not come without cost:

- So that the Inner Space protocol is immune to option stripping, it flags its presence using a magic number within the TCP Data of the initial segment in each direction, not a conventional TCP option in the header. This introduces a risk that payload in an ordinary SYN or SYN/ACK might be mistaken for the Inner Space protocol (an initial worst-case estimate of the probability is one connection globally every 40 years). Nonetheless, the risk is zero in the (currently common) case of an ordinary connection without payload

during the handshake. There is also no risk of a mistake the other way round--an upgraded connection cannot be mistaken for an ordinary connection.

- Although the dual handshake introduces no extra latency, it introduces extra connection processing & state, extra traffic and extra header processing. Initial estimates put the percentage overhead in single digits for connection processing and state, and traffic overhead at only a few hundredths of a percent. Nonetheless, once the most popular TCP servers have upgraded, only a single handshake will be necessary most of the time and overhead should drop to vanishingly small proportions.

Finally, it should be noted that the ambition of this work is more than just an incrementally deployable, low latency way to extend TCP option space. The aim is to move towards a more structured way for middleboxes to interact transparently with, rather than arbitrarily interfere with, end-system TCP stacks. This has been achieved for connection and stream control options, but it will still be hard to introduce new per-segment control options, which will still have to be located within the traditional Outer TCP Options.

1.1. Motivation for Adoption Now (to be removed before publication)

It seems inevitable that ultimately more option space will be needed, particularly given that many of the TCP options introduced recently consume large numbers of bits in order to provide sufficient information entropy, which is not amenable to compression.

Extension of TCP option space requires support from both ends. This means it will take many years before the facility is functional for most pairs of end-points. Therefore, given the problem is already becoming pressing, a solution needs to start being deployed now.

1.2. Scope

This experimental specification extends the TCP wire protocol. It is independent of the dynamic rate control behaviour of TCP and it is independent of (and thus compatible with) any protocol that encapsulates TCP, including IPv4 and IPv6.

1.3. Experiment Goals

TCP is critical to the robust functioning of the Internet, therefore any proposed modifications to TCP need to be thoroughly tested.

Success criteria: The experimental protocol will be considered successful if it satisfies the following requirements in the

consensus opinion of the IETF tcpm working group. The protocol needs to be sufficiently well specified so that more than one implementation can be built in order to test its function, robustness, overhead and interoperability (with itself, with previous version of TCP, and with various commonly deployed middleboxes). Non-functional issues such as recommendations on message timing also need to be tested. Various optional extensions to the protocol are proposed in Appendix A so experiments are also needed to determine whether these extensions ought to remain optional, or perhaps be removed or become mandatory.

Duration: To be credible, the experiment will need to last at least 12 months from publication of the present specification. If successful, it would then be appropriate to progress to a standards track specification, complemented by a report on the experiments.

1.4. Document Roadmap

The body of the document starts with a full specification of the Inner Space extension to TCP (Section 2). It is rather terse, answering 'What?' and 'How?' questions, but deferring 'Why?' to Section 3. The careful design choices made are not necessarily apparent from a superficial read of the specification, so the Design Rationale section is fairly extensive. The body of the document ends with Section 4 that checks possible interactions between the new scheme and pre-existing variants of TCP, including interaction with partial implementations of TCP in known middleboxes.

Appendix A specifies optional extensions to the protocol that will need to be implemented experimentally to determine whether they are useful. And Appendix B discusses the merits of the chosen design against alternative schemes.

1.5. 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]. 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.

TCP Header: As defined in [RFC0793]. Even though the present specification places TCP options beyond the Data Offset, the term 'TCP Header' is still used to mean only those fields at the head of the segment, delimited by the TCP Data Offset.

Inner TCP Options (or just Inner Options): TCP options placed in the space that the present specification makes available beyond the Data Offset.

Outer TCP Options (or just Outer Options): The TCP options in the traditional location directly after the base TCP Header and before the TCP Data Offset.

Prefix TCP Options: Inner Options to be processed before the Outer Options.

Suffix TCP Options: Inner Options to be processed after the Outer Options.

TCP options: Any TCP options, whether inner, outer or both. This specification makes this term on its own ambiguous so it should be qualified if it is intended to mean TCP options in a certain location.

TCP Payload: Data to be passed to the layer above TCP. The present specification redefines the TCP Payload so that it does not include the Inner TCP Options, the Inner Space Option and any Magic Number, even though they are located beyond the Data Offset.

TCP Data: The information in a TCP segment after the Data Offset, including the TCP Payload, Inner TCP Options, the Inner Space Option and the Magic Number defined in the present specification.

client: The process taking the role of actively opening a TCP connection.

server: The process taking the role of TCP listener.

Upgraded Segment: A segment that will only be fully understood by a host complying with the present specification (even though it might appear valid to a pre-existing TCP receiver). Similarly, Upgraded SYN, Upgraded SYN/ACK etc.

Ordinary Segment: A segment complying with pre-existing TCP specifications but not the present specification. Similarly, Ordinary SYN, Ordinary SYN/ACK etc.

Upgraded Connection: A connection starting with an Upgraded SYN.

Ordinary Connection: A connection starting with an Ordinary SYN.

Upgraded Host: A host complying with the present document as well as with pre-existing TCP specifications. Similarly Upgraded TCP Client, Upgraded TCP Server, etc.

Legacy Host: A host complying with pre-existing TCP specifications, but not with the present document. Similarly Legacy TCP Client, Legacy TCP Server, etc.

Note that the term 'Ordinary' is used for segments and connections, but the term 'Legacy' is used for hosts. This is because, if the Inner Space protocol were widely used in future, a host that could not open an Upgraded Connection would be considered deficient and therefore 'Legacy', whereas an Ordinary Connection would not be considered deficient in the future; because it will always be legitimate to open an Ordinary Connection if extra option space is not needed.

2. Protocol Specification

2.1. Protocol Interaction Model

2.1.1. Dual 3-Way Handshake

During initial deployment, an Upgraded TCP Client sends two alternative SYNs: an Ordinary SYN in case the server is legacy and a SYN-U in case the server is upgraded. The two SYNs MUST have the same network addresses and the same destination port, but different source ports. Once the client establishes which type of server has responded, it continues the connection appropriate to that server type and aborts the other without completing the 3-way handshake.

The format of the SYN-U will be described later (Section 2.2.2). At this stage it is only necessary to know that the client can put either TCP options or payload (or both) in a SYN-U, in the space traditionally intended only for payload. So if the server's response shows that it does not recognise the Upgraded SYN-U, the client is responsible for aborting the Upgraded Connection. This ensures that a Legacy TCP Server will never erroneously confuse the application by passing it TCP options as if they were user-data.

Section 3.1 explains various strategies the client can use to send the SYN-U first and defer or avoid sending the Ordinary SYN. However, such strategies are local optimizations that do not need to be standardized. The rules below cover the most aggressive case, in which the client sends the SYN-U then the Ordinary SYN back-to-back to avoid any extra delay. Nonetheless, the rules are just as applicable if the client defers or avoids sending the Ordinary SYN.

Table 1 summarises the TCP 3-way handshake exchange for each of the two SYNs in the two right-hand columns, between an Upgraded TCP Client (the active opener) and either:

1. a Legacy Server, in the top half of the table (steps 2-4), or

2. an Upgraded Server, in the bottom half of the table (steps 2-4)

Because the two SYNs come from different source ports, the server will treat them as separate connections, probably using separate threads (assuming a threaded server). A load balancer might forward each SYN to separate replicas of the same logical server. Each replica will deal with each incoming SYN independently - it does not need to co-ordinate with the other replica.

		Ordinary Connection	Upgraded Connection
1	Upgraded Client	>SYN	>SYN-U
/\/\	/\/\/\/\/\/\/\	/\/\/\/\/\/\/\/\	/\/\/\/\/\/\/\/\
2	Legacy Server	<SYN/ACK	<SYN/ACK
3a	Upgraded Client	Waits for response to both SYNs	
3b	"	>ACK	>RST
4		Cont...	
/\/\	/\/\/\/\/\/\/\	/\/\/\/\/\/\/\/\	/\/\/\/\/\/\/\/\
2	Upgraded Server	<SYN/ACK	<SYN/ACK-U
3a	Upgraded Client	Waits for response to SYN-U	
3b	"	>RST	>ACK
4			Cont...

Table 1: Dual 3-Way Handshake in Two Server Scenarios

Each column of the table shows the required 3-way handshake exchange within each connection, using the following symbols:

> means client to server;

< means server to client;

Cont... means the TCP connection continues.

The connection that starts with an Ordinary SYN is called the 'Ordinary Connection' and the one that starts with a SYN-U is called the 'Upgraded Connection'. An Upgraded Server MUST respond to a SYN-U with an Upgraded SYN/ACK (termed a SYN/ACK-U and defined in Section 2.2.2). Then the client recognises that it is talking to an Upgraded Server. The client's behaviour depends on which response it receives first, as follows:

- If the client first receives a SYN/ACK response on the Ordinary Connection, it MUST wait for the response on the Upgraded Connection. It then proceeds as follows:
  - If the response on the Upgraded Connection is an Ordinary SYN/ACK, the client MUST reset (RST) the Upgraded Connection and it can continue with the Ordinary Connection.
  - If the response on the Upgraded Connection is an Upgraded SYN/ACK-U, the client MUST reset (RST) the Ordinary Connection and it can continue with the Upgraded Connection.
- If the client first receives an Ordinary SYN/ACK response on the Upgraded Connection, it MUST reset (RST) the Upgraded Connection immediately. It can then wait for the response on the Ordinary Connection and, once it arrives, continue as normal.
- If the client first receives an Upgraded SYN/ACK-U response on the Upgraded Connection, it MUST reset (RST) the Ordinary Connection immediately and continue with the Upgraded Connection.

2.1.2. Dual Handshake Retransmission Behaviour

If the client receives a response to the SYN, but a short while after that {ToDo: duration TBA} the response to the SYN-U has not arrived, it SHOULD retransmit the SYN-U. If latency is more important than the extra TCP option space, in parallel to any retransmission, or instead of any retransmission, the client MAY give up on the Upgraded (SYN-U) Connection by sending a reset (RST) and completing the 3-way handshake of the Ordinary Connection.

If the client receives no response at all to either the SYN or the SYN-U, it SHOULD solely retransmit one or the other, not both. If latency is more important than the extra TCP option space, it will retransmit the SYN. Otherwise it will retransmit the SYN-U. It MUST

NOT retransmit both segments, because the lack of response could be due to severe congestion.

2.1.3. Continuing the Upgraded Connection

Once an Upgraded Connection has been successfully negotiated in the SYN, SYN/ACK exchange, either host can allocate any amount of the TCP Data space in any subsequent segment for extra TCP options. In fact, the sender has to use the upgraded segment structure in every subsequent segment of the connection that contains non-zero TCP Payload. The sender can use the upgraded structure in a segment carrying no user-data (e.g. a pure ACK), but it does not have to.

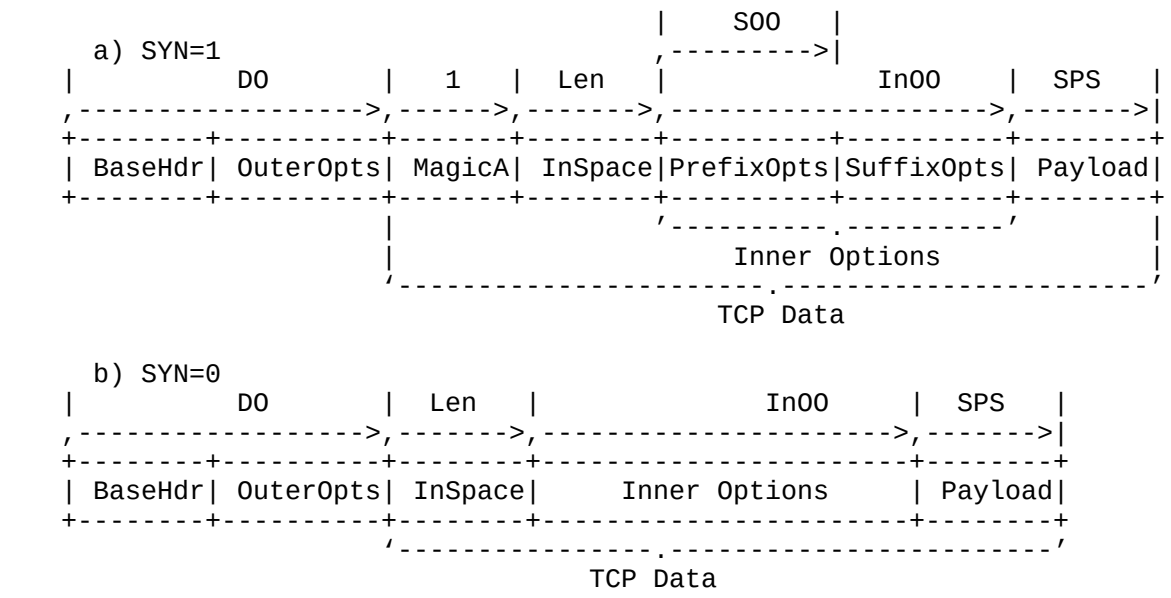
As well as extra option space, the facility offers other advantages, such as reliable ordered delivery of Inner TCP Options on empty segments and more robust middlebox traversal. If none of these features is needed, at any point the facility can be disabled for the rest of the connection, using the ModeSwitch TCP option in Appendix A.1. Interestingly, the ModeSwitch options itself can be very simple because it uses the reliable ordered delivery property of Inner Options, rather than having to cater for the possibility that a message to switch to disabled mode might be lost or reordered.

2.2. Upgraded Segment Structure and Format

2.2.1. Structure of an Upgraded Segment

An Upgraded Segment is structured as shown in Figure 2. Up to the TCP Data Offset, the structure is identical to an Ordinary TCP Segment, with a base TCP Header (BaseHdr) and the usual facility to set the Data Offset (DO) to allow space for TCP options. These regular TCP options are renamed by this specification to Outer TCP Options or just Outer Options, and labelled as OuterOpts in the figure.

The first segment in each direction (i.e. the SYN or the SYN/ACK) is identifiable as upgraded by the presence of the 4-octet Magic Number A (MagicA) at the start of the TCP Data. The probability that an Upgraded Server will mistake arbitrary data at the beginning of the payload of an Ordinary Segment for the Magic Number has to be allowed for, but it is vanishingly small (see Section 3.2.1). Once an Upgraded Connection has been negotiated during the SYN - SYN/ACK exchange, a magic number is not needed to identify Upgraded Segments, because both ends know that the protocol requires the sender to use the upgraded format on all subsequent segments with non-zero TCP Data. Aside from the magic number, the structure of the rest of an Upgraded Segment is effectively the same whether a) SYN=1 or b) SYN=0.



All offsets are specified in 4-octet (32-bit) words, except SPS, which is in octets.

Figure 2: The Structure of an Upgraded Segment (not to scale)

Unlike an Ordinary TCP Segment, the Payload of an Upgraded Segment does not start straight after the TCP Data Offset. Instead, Figure 2 shows that space is provided for additional Inner TCP Options before the TCP Payload. The size of this space is termed the Inner Options Offset (InOO). The TCP receiver reads the InOO field from the Inner Option Space (InSpace) option defined in Section 2.2.2.

The InSpace Option is located in a standardized location so that the receiver can find it:

- On a segment with SYN=1, an Upgraded TCP Sender MUST locate the InSpace Option straight after the magic number, specifically 4 * (DO + 1) octets from the start of the segment.

- On a segment with SYN=0, an Upgraded TCP Sender MUST locate the InSpace Option at the beginning of the TCP Data, specifically 4 * DO octets from the start of the segment.

Because the InSpace Option is only ever located in a standardized location it does not need to follow the RFC 793 format of a TCP option. Therefore, although we call InSpace an 'option', we do not describe it as a 'TCP option'.

The Sent Payload Size (SPS) is also read from within the InSpace Option. If the byte-stream has been resegmented, it allows the receiver to step from one InSpace Option to the next even if the InSpace Options are no longer at the start of each segment (see Section 2.3).

On a segment with SYN=1 (i.e. a SYN or SYN/ACK) the Suffix Options Offset (SOO) is also read from within the InSpace Option. It delineates the end of the Prefix TCP Options (PrefixOpts in the figure) and the start of the Suffix TCP Options (SuffixOpts). When SYN=1, the receiver processes PrefixOpts before OuterOpts, then SuffixOpts afterwards. When SYN=0, the receiver processes the Outer Options before the Inner Options. Full details of option processing are given in Section 2.3.

2.2.2. Format of the InSpace Option

The internal structure of the InSpace Option for an Upgraded SYN or SYN/ACK segment (SYN=1) is defined in Figure 3a) and for a segment with SYN=0 in Figure 3b).

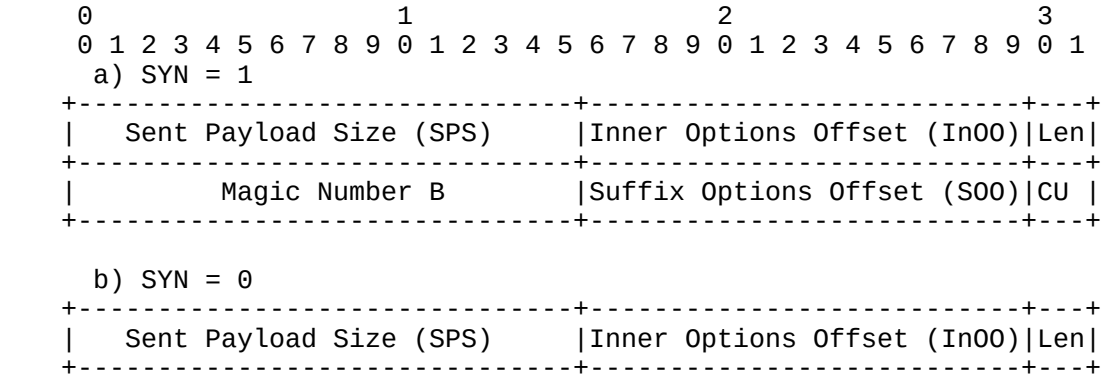


Figure 3: InSpace Option Format

The fields are defined as follows (see Section 3.3 for the rationale behind these format choices):

Option Length (Len): The 2-bit Len field specifies the length of the InSpace Option in 4-octets words (see Section 3.3 for rationale). For this experimental specification:

When SYN=1: the sender MUST use Len=2;

When SYN=0: the sender MUST use Len=1.

Sent Payload Size (SPS): In this 16-bit field the sender MUST record the size in octets of the TCP Payload when it was sent. This specification defines the TCP Payload as solely the user-data to be passed to the application. This excludes Inner TCP options, the InSpace Option and any magic number.

Inner Options Offset (InOO): This 14-bit field defines the total size of the Inner TCP Options in 4-octet words.

The following fields are only defined on a segment with SYN=1 (i.e. a SYN or SYN/ACK):

Magic Number B: The sender MUST fill this 16-bit field with Magic Number B {ToDo: Value TBA} to further reduce the chance that a receiver will mistake the end of an arbitrary Ordinary Payload for the InSpace Option.

Suffix Options Offset (SOO): The 14-bit SOO field defines an additional offset in 4-octet words from the start of the Inner Options that identifies the extent of the Prefix Options (see Section 2.3.2).

Currently Unused (CU): The sender MUST fill the CU field with zeros and they MUST be ignored and forwarded unchanged by other nodes, even if their value is different.

2.3. Inner TCP Option Processing

2.3.1. Writing Inner TCP Options

2.3.1.1. Constraints on TCP Fast Open

If an Upgraded TCP Client uses a TCP Fast Open (TFO) cookie [I-D.ietf-tcpm-fastopen] in an Upgraded SYN-U, it MUST place the TFO option within the Inner TCP Options, beyond the Data Offset.

This rule is specific to TFO, but it can be generalised to any capability similar to TFO as follows: An Upgraded TCP Client MUST NOT place any TCP option in the Outer TCP Options of a SYN if it might cause a TCP server to pass user-data directly to the application before its own 3-way handshake completes.

If a client uses TCP Fast Open cookies on both the parallel connection attempts of a dual handshake, an Upgraded Server will deliver the TCP Payload to the application twice before the client aborts the Ordinary Connection. This is not a problem, because [I-D.ietf-tcpm-fastopen] requires that TFO is only used for applications that are robust to duplicate requests.

2.3.1.2. Option Alignment

If the end of the last Inner TCP Option does not align on a 4-octet boundary, the sender MUST append sufficient no-op TCP options. On a SYN=1 segment, the end of the Prefix TCP Options MUST be similarly aligned.

If a block-mode transformation (e.g. compression or encryption) is being used, the sender might have to add some padding options to align the end of the Inner Options with the end of a block. Any future encryption specification will need to carefully define this padding in order not to weaken the cipher.

2.3.1.3. Sequence Space Coverage

TCP's sequence number and acknowledgement number space MUST include all the TCP Data, i.e. the InSpace Option, any Inner Options, and any magic number as well as the TCP Payload. Similarly, the sender MUST NOT transmit any form of TCP Data unless the advertised receive window is sufficient. These rules have significant implications, which are discussed in Section 3.2.4.

2.3.1.4. Presence or Absence of Payload

Whenever the sender includes non-zero user-data payload in a segment, it MUST also include an InSpace Option, whether or not there are any Inner Options.

If the sender includes no user-data in a segment (e.g. pure ACKs, RSTs) it MAY include an InSpace Option but it does not have to. {ToDo: Consider whether there is any reason to preclude Inner Options on a RST, FIN or FIN-ACK.}

Once a sender has included the InSpace Option and possibly other Inner Options on a segment with no TCP Payload, while it has no further user-data to send it SHOULD NOT repeat the same set of control options on subsequent segments. Thus, in a sequence of pure ACKs, any particular set of Inner Options will only appear once, and other pure ACKs will be empty. The only envisaged exception to this rule would be infrequent repetition (i.e. tens of minutes to hours) of the same control options, which might be necessary to provide a heartbeat or keep-alive capability.

2.3.2. Reading Inner TCP Options

The rules for reading Inner TCP Options are divided between the following two subsections, depending on whether SYN=1 or SYN=0.

2.3.2.1. Reading Inner TCP Options (SYN=1)

This subsection applies when TCP receives a segment with SYN=1, i.e. when the server receives a SYN or the client receives a SYN/ACK.

Before processing any TCP options, unless the size of the TCP Data is less than 8 octets, an Upgraded Receiver MUST determine whether the segment is an Upgraded Segment by checking that all the following conditions apply:

- The first 4 octets of the segment match Magic Number A;
- The value of the Length field of the InSpace Option is 2;
- The value of Magic Number B in the InSpace Option is correct;
- The value of the Sent Payload Size matches the size of the TCP Payload.

If all these conditions pass, the receiver MAY walk the sequence of Inner TCP Options, using the length of each to check that the sum of their lengths equals InOO. The receiver then concludes that the received segment is an Upgraded Segment.

The receiver then processes the TCP Options in the following order:

1. Any Prefix TCP options (PrefixOpts in Figure 2)
2. Any Outer TCP options (OuterOpts in Figure 2);
3. Any Suffix TCP options (SuffixOpts in Figure 2)

The receiver removes the magic number, the InSpace Option and each TCP Option from the TCP Data as it processes each. This frees up receive buffer, so the receiver increases its local value of the receive window accordingly. Once only the TCP Payload remains, the receiver holds it ready to pass to the application. It then returns the appropriate Upgraded Acknowledgement to progress the dual handshake (see Section 2.1.1).

If any of the above tests to find the InSpace Option fails:

1. the receiver concludes that the received segment is an Ordinary Segment. It MUST then proceed by processing any Outer TCP options in the TCP Header in the normal order (OuterOpts in Figure 2).

2. If some previous control message causes the TCP receiver to alter the TCP Data (e.g. decompression, decryption), it reruns the above tests to check if the altered TCP Data now looks like an Upgraded Segment.

3. If it finds an InSpace Option, it suspends processing the Outer TCP Options and instead processes and removes TCP Options in the following order:

   1. Any Prefix Inner Options;

   2. Any remaining Outer TCP Options;

   3. Any Suffix Inner Options.

4. If it does not find an InSpace Option, it continues processing the remaining Outer TCP Options as normal.

For the avoidance of doubt the above rules imply that, as long as an InSpace Option has not been found in the segment, the receiver might rerun the tests for it multiple times if multiple Outer TCP Options alter the TCP Data. However, once the receiver has found an InSpace Option, it MUST NOT rerun the tests for an Upgraded Segment in the same segment.

If the receiver has not found an InSpace Option after processing all the Outer Options, it returns the appropriate Ordinary Acknowledgement to progress the dual handshake (see Section 2.1.1). As normal, it holds any TCP Payload ready to pass to the application.

2.3.2.2. Reading Inner TCP Options (SYN=0)

This subsection applies once the TCP connection has successfully negotiated to use the upgraded InSpace structure.

As each segment with SYN=0 arrives, the receiver immediately processes any Outer TCP options.

As the receiver buffers TCP Data, it uses TCP’s regular mechanisms to fill any gaps due to reordering or loss so that it can work its way along the ordered byte-stream. As the receiver encounters each set of Inner Options, it MUST process them in the order they were sent, as illustrated in Figure 4a) in Section 3.2.4. The receiver MUST remove the InSpace Option and Inner TCP Options from the TCP Data as it processes them, adding to the receive window accordingly. Once only the TCP Payload remains the receiver passes it to the application.

It uses each InSpace Option to calculate the extent of the associated Inner Options (using InOO), and the amount of payload data before the next InSpace Option (using Sent Payload Size). The receiver MUST NOT locate InSpace Options by assuming there is one at the start of the TCP Data in every segment, because resegmentation might invalidate this assumption.

Therefore, the receiver processes the Inner Options in the order they were sent, which is not necessarily the order in which they are received. And if an Inner Option applies to the data stream, the receiver applies it at the point in the data stream where the sender inserted it. As a consequence, the receiver always processes the Inner Options after the Outer Options.

The Inner Options are deliberately placed within the byte-stream so that the sender can transform them along with the payload data, e.g. to compress or encrypt them. A previous control message might have required the TCP receiver to alter the byte-stream before passing it to the application, e.g. decompression or decryption. If so, the TCP receiver applies transformations progressively, to one sent segment at a time, in the following order:

1. The receiver MUST apply any transformations to the byte-stream up to the end of the next set of Inner Options, i.e. over the extent of the next Sent Payload Size, InSpace Option and any Inner Options.

2. The receiver MUST then process and remove the InSpace Option and any Inner Options (which might change the way it transforms the next segment, e.g. a rekey option).

3. Having established the extent of the next sent segment, The receiver returns to step 1.

2.3.3. Forwarding Inner TCP Options

Middleboxes exist that process some aspects of the TCP Header. Although the present specification defines a new location for Inner TCP Options beyond the Data Offset, this is intended for the exclusive use of the destination TCP implementation. Therefore:

- A middlebox MUST treat any octets beyond the Data Offset as immutable user-data. Legacy Middleboxes already do not expect to find options beyond the Data Offset anyway.

- A middlebox MUST NOT defer data in a segment with SYN=1 to a subsequent segment.

A TCP implementation is not necessarily aware whether it is deployed in a middlebox or in a destination, e.g. a split TCP connection might use a regular off-the-shelf TCP implementation. Therefore, a general-purpose TCP that implements the present specification will need a configuration switch to disable any search for options beyond the Data Offset and to enable immediate forwarding of data in a SYN.

2.4. Exceptions

{ToDo: Define behaviour of forwarding or receiving nodes if the structure or format of an Upgraded Segment is not as specified.}

If an Upgraded TCP Receiver receives an InSpace Option with a Length it does not recognise as valid, it MUST drop the packet and acknowledge the octets up to the start of the unrecognised option.

Values of Sent Payload Size greater than 2^16 - 25 (=65,511) octets in a regular (non-jumbo) InSpace Option MUST be treated as the distance to the next InSpace option, but they MUST NOT be taken as indicative of the size of the TCP Payload when it was sent. This is because the TCP Payload in a regular IPv6 packet cannot be greater than (2^16 -1 - 20 - 4) octets (given the minimum TCP header is 20 octets and the minimum InSpace Option is 4 octets). A Sent Payload Size of 0xFFFF octets MAY be used to minimise the occurrence of empty InSpace options without permanently disabling the Inner Space protocol for the rest of the connection.

If the size of the payload is greater than 65,511 octets, the sender MUST use a Jumbo InSpace Option (Appendix A.3).

2.5. SYN Flood Protection

An implementation of the Inner Space protocol MUST support the EchoCookie TCP option [I-D.briscoe-tcpm-echo-cookie]. To indicate its support for EchoCookie, an Ordinary Client would send an empty EchoCookie TCP option on the SYN. Support for the Inner Space protocol makes this redundant. Therefore an Inner Space client MUST NOT send an empty EchoCookie TCP option on a SYN-U.

The EchoCookie TCP option replaces the SYN Cookie mechanism [RFC4987], which only has sufficient space to hold the result of one TCP option negotiation (the MSS), and then only a subset of the possible values (see the discussion under Security Considerations Section 6).

3. Design Rationale

This section is informative, not normative.

3.1. Dual Handshake and Migration to Single Handshake

In traditional [RFC0793] TCP, the space for options is limited to 40B by the maximum possible Data Offset. Before a TCP sender places options beyond that, it has to be sure that the receiver will understand the upgraded protocol, otherwise it will confuse and potentially crash the application by passing it TCP options as if they were payload data.

The Dual Handshake (Section 2.1.1) ensures that a Legacy TCP Server will never pass on TCP options as if they were user-data. If a SYN carries TCP Data, a TCP server typically holds it back from the application until the 3-way handshake completes. This gives the client the opportunity to abort the Upgraded Connection if the response from the server shows it does not recognise an Upgraded SYN.

The strategy of sending two SYNs in parallel is not essential to the Alternative SYN approach. It is merely an initial strategy that minimises latency when the client does not know whether the server has been upgraded. Evolution to a single SYN with greater option space could proceed as follows:

- Clients could maintain a white-list of upgraded servers discovered by experience and send just the Upgraded SYN-U in these cases.

- Then, for white-listed servers, the client could send an Ordinary SYN only in the rare cases when an attempt to use an Upgraded Connection had previously failed (perhaps a mobile client encountering a new blockage on a new path to a server that it had previously accessed over a good path).

- In the longer term, once it can be assumed that most servers are upgraded and the risk of having to fall back to legacy has dropped to near-zero, clients could send just the Upgraded SYN first, without maintaining a white-list, but still be prepared to send an Ordinary SYN in the rare cases when that might fail.

There is concern that, although dual handshake approaches might well eventually migrate to a single handshake, they do not scale when there are numerous choices to be made simultaneously. For instance:

- trying IPv6 then IPv4 [RFC6555];

- o and trying SCTP and TCP in parallel [I-D.wing-tsvwg-happy-eyeballs-sctp];
- o and trying ECN and non-ECN in parallel;
- o and so on.

Nonetheless, it is not necessary to try every possible combination of N choices, which would otherwise require 2^N handshakes (assuming each choice is between two options). Instead, a selection of the choices could be attempted together. At the extreme, two handshakes could be attempted, one with all the new features, and one without all the new features.

3.2. In-Band Inner Option Space

3.2.1. Non-Deterministic Magic Number Approach

This section justifies the magic number approach by contrasting it with a more 'conventional' approach. A conventional approach would use a regular (Outer) TCP option to point to the dividing line within the TCP Data between the extra Inner Options and the TCP Payload.

This 'conventional' approach cannot provide extra option space over a path on which a middlebox strips TCP options that it does not recognise. [Honda11] quantifies the prevalence of such paths. It reports on experiments conducted in 2010-2011 that found unknown options were stripped from the SYN-SYN/ACK exchange on 14% of paths to port 80 (HTTP), 6% of paths to port 443 (HTTPS) and 4% of paths to port 34343 (unassigned). Further analysis found that the option-stripping middleboxes fell into two main categories:

- o about a quarter appeared to actively remove options that they did not recognise (perhaps assuming they might be indicative of an attack?);
- o the rest were some type of higher layer proxy that split the TCP connection, unwittingly failing to pass unknown options between the two connections.

In contrast, the magic number approach ensures that not only are the Inner Options tucked away beyond the Data Offset, but the option that gives the extent of the Inner Options is also beyond the Data Offset (see Section 2.2.1). This ensures that all the TCP Headers and options up to the Data Offset are completely indistinguishable from an Ordinary Segment. It is very unusual for a middlebox not to forward TCP Data unchanged, so it will be highly likely (but not certain--see Appendix A.2.4) to forward the extra Inner Options.

The downside of the magic number approach is that it is slightly non-deterministic, quantified as follows:

- The probability that an Upgraded SYN=1 segment will be mistaken for an Ordinary Segment is precisely zero.

- In the currently common case of a SYN with zero payload, the probability that it will be mistaken for an Upgraded Segment is also precisely zero.

- However, there will be a very small probability (roughly 2^{-66} or 1 in 74 billion billion (74 * 10^18)) that payload data in an Ordinary SYN=1 segment could be mistaken for an Upgraded SYN or SYN/ACK, if it happens to contain a pattern in exactly the right place that matches the correct Sent Payload Size, Length and Magic Numbers of an InSpace Option. {ToDo: Estimate how often a collision will occur globally. Rough estimate: 1 connection collision globally every 40 years.}

The above probability is based on the assumptions that:

- the magic numbers will be chosen randomly (in reality they will not--for instance, a magic number that looked just like the start of an HTTP connection would be rejected)

- data at the start of Ordinary SYN=1 segments is random (in reality it is not--the first few bytes of most payloads are very predictable).

Therefore even though 2^{-66} is a vanishingly small probability, the actual probability of a collision will be much lower.

If a collision does occur, it will result in TCP removing a number of 32-bit words of data from the start of a byte-stream before passing it to the application.

3.2.2. Non-Goal: Security Middlebox Evasion

The purpose of locating control options within the TCP Data is not to evade security. Security middleboxes can be expected to evolve to examine control options in the new inner location. Instead, the purpose is to traverse middleboxes that block new TCP options unintentionally--as a side effect of their main purpose--merely because their designers were too careless to consider that TCP might evolve. This category of middleboxes tends to forward the TCP Payload unaltered.

By sitting within the TCP Data, the Inner Space protocol should traverse enough existing middleboxes to reach critical mass and prove itself useful. In turn, this will open an opportunity to introduce integrity protection for the TCP Data (which includes Inner Options). Whereas today, no operating system would introduce integrity protection of Outer TCP options, because in too many cases it would fail and abort the connection. Once the integrity of Inner Options is protected, it will raise the stakes. Any attempt to meddle with control options within the TCP Data will not just close off the theoretical potential benefit of a protocol advance that no-one knows they want yet; it will fail integrity checks and therefore completely break any communication. It is unlikely that a network operator will buy a middlebox that does that.

Then middlebox designers will be on the back foot. To completely block communications they will need a sound justification. If they block an attack, that will be fine. But if they want to block everything abnormal, they will have to block the whole communication, or nothing. So the operator will want to choose middlebox vendors who take much more care to ensure their policies track the latest protocol advances--to avoid costly support calls.

3.2.3. Avoiding the Start of the First Two Segments

Some middleboxes discard a segment sent to a well-known port (particularly port 80) if the TCP Data does not conform to the expected app-layer protocol (particularly HTTP). Often such middleboxes only parse the start of the app-layer header (e.g. Web filters only continue until they find the URL being accessed, or DPI boxes only continue until they have identified the application-layer protocol).

The segment structure defined in Section 2.2.1 would not traverse such middleboxes. An alternative segment structure that avoids the start of the first two segments in each direction is defined in Appendix A.4. It is not mandatory to implement in the present specification. However, it is hoped that it will be included in some experimental implementations so that it can be decided whether it is worth making mandatory.

3.2.4. Control Options Within Data Sequence Space

Including Inner Options within TCP’s sequence space gives the sender a simple way to ensure that control options will be delivered reliably and in order to the remote TCP, even if the control options are on segments without user-data. By using TCP’s existing stream delivery mechanisms, it adds no extra protocol processing, no extra packets and no extra bits.

The sender can even choose to place control options on a segment without user-data, e.g. to reliably re-key TCP-level encryption on a connection currently sending no data in one direction. The sender can even add an InSpace Option without further Inner Options. Then it can ensure that the segment will automatically be delivered reliably and in order to the remote TCP, even though it carries no user-data or other TCP control options, e.g. for a test probe, a tail-loss probe or a keep-alive.

Figure 4a) illustrates control options arriving reliably and in order at the receiving TCP stack in comparison with the traditional approach shown in Figure 4b), in which control options are outside the sequence space. In the traditional approach, during a period when the remote TCP is sending no user-data, the local TCP may receive control options E, B and D without ever knowing that they are out of order, and without ever knowing that C is missing.

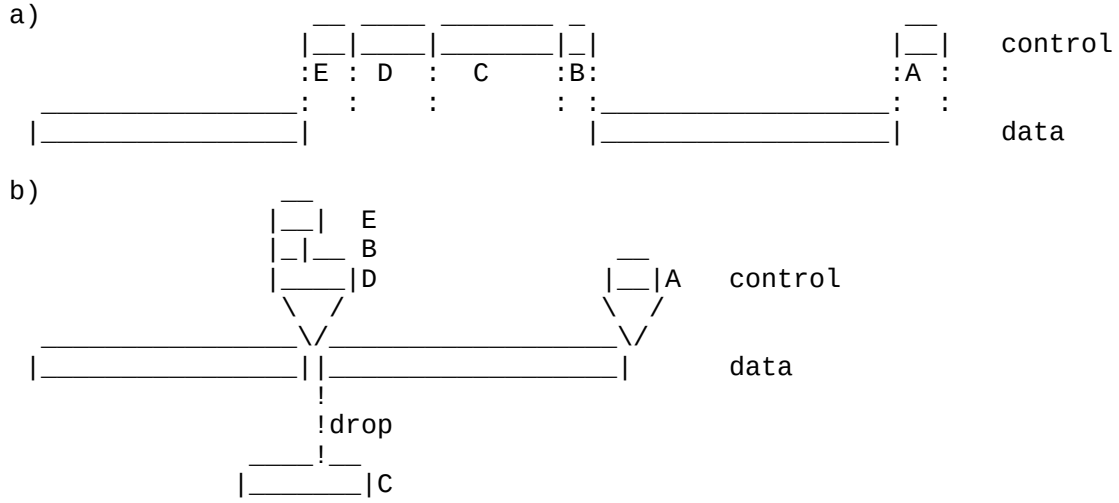


Figure 4: Control options a) inside vs. b) outside TCP sequence space‘

By including Inner Options within the sequence space, each control option is automatically bound to the start of a particular byte in the data stream, which makes it easy to switch behaviour at a specific point mid-stream (e.g. re-keying or switching to a different control mode). With traditional TCP options, a bespoke reliable and ordered binding to the data stream would have to be developed for each TCP option that needs this capability (e.g. co-ordinating use of new keys in TCP-AO [RFC5925] or tcpcrypt [I-D.bittau-tcpinc]).

Including Inner Options in sequence also allows the receiver to tell the sender the exact point at which it encountered an unrecognised TCP option using only TCP's pre-existing byte-granularity acknowledgement scheme.

Middleboxes exist that rewrite TCP sequence and acknowledgement numbers, and they also rewrite options that refer to sequence numbers (at least those known when the middlebox was produced, such as SACK, but not any introduced afterwards). If Inner Options were not included in sequence, the number of bytes beyond the TCP Data Offset in each segment would not match the sequence number increment between segments. Then, such middleboxes could unintentionally corrupt the user-data and options by 'normalising' sequence or acknowledgement numbering. Fortunately, including Inner Options in sequence improves robustness against such middleboxes.

3.2.5. Rationale for the Sent Payload Size Field

A middlebox that splits a TCP connection can coalesce and/or divide the original segments. Segmentation offload hardware introduces similar resegmentation. Inclusion of the Sent Payload Size field in the InSpace Option makes the scheme robust against such resegmentation.

The Sent Payload Size is not strictly necessary on a SYN (SYN=1, ACK=0) because a SYN is never resegmented. However, for simplicity, the layout for a SYN is made the same as for a SYN/ACK. This future-proofs the protocol against the possibility that SYNs might be resegmented in future. And it makes it easy to introduce the alternative segment structure of Appendix A.4 if it is needed.

3.3. Rationale for the InSpace Option Format

The format of the InSpace Option (Figure 3) does not necessarily have to comply with the RFC 793 format for TCP options, because it is not intended to ever appear in a sequence of TCP options. In particular, it does not need an Option Kind, because the option is always in a known location. In effect the magic number serves as a multi-octet Option Kind for the first InSpace Option, and the location of each subsequent options is always known as an offset from the previous one, using InOO and Sent Payload Size fields.

Other aspects of the layout are justified as follows:

Length: Whatever the size of the InSpace Option, the right-hand edge of the Length field is always located 4 octets from the start of the option, so that the receiver can find it to determine the layout of the rest of the option. The option is always a multiple

of 4 octets long, so that any subsequent Inner TCP Options comply with TCP's option alignment requirements.

Sent Payload Size: This field is 16 bits wide, which is reasonable given segment size cannot exceed the limits set by the Total Length field in the IPv4 header and the Payload Length field in the IPv6 header, both of which are 16 bits wide.

If the sender were to use a jumbogram [RFC2675], it could use the Jumbo InSpace Option defined in Appendix A.3, which offers a 32-bit Sent Payload Size field. The Jumbo InSpace Option is not mandatory to implement for the present experimental specification. Even if it is implemented, it is only defined when SYN=0, given use of a jumbogram for a SYN or SYN/ACK would significantly exceed other limits that TCP sets for these segments.

InSpace Options Offset The 14-bit field is in units of 4-octet words, in order to restrict Inner Options to no less than the size of a maximum sized segment (given 4 * 2^14 = 2^16 octets).

When SYN=1 the layout of the InSpace Option is extended to include:

Suffix Options Offset: The SOO field is the same 14-bit width as the InOO field, and for the same reason. Both the SOO and InOO fields are aligned 2 bits to the left of a word boundary so that they can be used directly in units of octets by masking out the 2-bit field to the right.

Magic Number B: The 32-bit size of Magic Number A is not enough to reduce the probability of mistaking the start of an Ordinary SYN Payload for the start of the Inner Space protocol. A 64-bit magic number could have been provided by using the next 4-octet word, but this would be unnecessarily large. Therefore, when SYN=1, 16 more bits of magic number are provided within the InSpace Option. Otherwise, these 16-bits would only have to be used for padding to align with the next 4-octet word boundary anyway.

3.4. Protocol Overhead

The overhead of the Inner Space protocol is quantified as follows:

Dual Handshake:

Latency:

Upgraded Server : zero;

Legacy Server: worst latency of the dual handshakes.

Connection Rate: The typical connection rate will inflate by P*D, where:

P [0-100%] is the proportion of connections that use extra option space;

D [0-100%] is the proportion of these that use a dual handshake (the remainder use a single handshake, e.g. by caching knowledge of upgraded servers).

For example, if P=80% and D=10%, the connection rate will inflate by 8%. P is difficult to predict. D is likely to be small, and in the longer term it should reduce to the proportion of connections to remaining legacy servers, which are likely to be the less frequently accessed ones. In the worst case if both P & D are 100%, the maximum that the connection rate can inflate by is 100% (i.e. to twice present levels).

Connection State: Connection state on servers and middleboxes will inflate by P*D/R, where

R is the average hold time of connection state measured in round trip times

This is because a server or middlebox only holds dual connection state for one round trip, until the RST on one of the two connections. For example, keeping P & D as they were in the above example, if R = 3 round trips {ToDo: TBA}, connection state would inflate by 2.7%. In the longer term, any extra connection state would be focused on legacy servers, with none on upgraded servers. Therefore, if memory for dual handshake flow state was a problem, upgrading the server to support the Inner Space protocol would solve the problem.

Network Traffic: The network traffic overhead is 2*H*P*D/J counting in bytes or 2*P*D/K counting in packets, where

H is 88B for IPv4 or 108B for IPv6 (assuming the Ordinary SYN and SYN/ACK have a TCP header packed to the maximum of 60B with TCP options, they have no TCP payload, their IP headers have no extensions and the InSpace Option in the SYN-U and SYN/ACK-U is 8B);

J is the average number of bytes per TCP connection (in both directions)

K is the average number of packets per TCP connection (in both directions);

For example, keeping and P & D as they were in the above example, if J = 50KiB for IPv4 and K = 70 packets (ToDo: TBA), traffic overhead would be 0.03% counting in bytes or 0.2% counting in packets.

Processing: {ToDo: Implementation tests}

InSpace Option on every non-empty SYN=0 segment:

Network Traffic: The traffic overhead is P*Q*4/F, where

Q is the proportion of Inner Space connections that leave the protocol enabled after the initial handshake;

F is the average frame size in bytes (assuming one segment per frame).

This is because the InSpace option adds 4B per segment. For example, keeping P as it was in the above example and taking Q=10% and F=750B, the traffic overhead is 0.04%. It is as difficult to predict Q as it is to predict P.

Processing: {ToDo: Implementation tests}

4. Interaction with Pre-Existing TCP Implementations

4.1. Compatibility with Pre-Existing TCP Variants

A TCP option MUST by default only be used as an Outer Option, unless it is explicitly specified that it can (or must) be used as an Inner Option. The following list of pre-existing TCP options can be located as Inner Options:

- Maximum Segment Size (MSS) [RFC0793];
- SACK-ok [RFC2018];
- Window Scale [RFC7323];
- Multipath TCP [RFC6824], except the Data ACK part of the Data Sequence Signal (DSS) option;
- TCP Fast Open [I-D.ietf-tcpm-fastopen];
- The tcpcrypt CRYPT option [I-D.bittau-tcpinc].

The following MUST NOT be located as Inner Options:

- Timestamp [RFC7323];
- SACK [RFC2018];
- The Data ACK part of the DSS option of Multipath TCP [RFC6824];
- TCP-AO [RFC5925];
- The tcpcrypt MAC option [I-D.bittau-tcpinc] as long as it covers the TCP header.

{ToDo: The above list is not authoritative. Many of the above schemes involve multiple different types of TCP option, and all the types need to be separately assessed.}

The Inner Space protocol supports TCP Fast Open, by constraining the client to obey the rules in Section 2.3.1.1).

All the sub-types of the MPTCP option [RFC6824] except one could be located as Inner Options. That is, MP_CAPABLE, MP_JOIN, ADD_ADDR(2), REMOVE_ADDR, MP_PRIO, MP_FAIL, MP_FASTCLOSE. The Data Sequence Signal (DSS) of MPTCP consists of four separable parts: i) the Data ACK; ii) the mapping between the Data Sequence Number and the Subflow Sequence Number over a Data-Level Length; iii) the Checksum; and iv) the DATA_FIN flag. If MPTCP were re-factored to take advantage of the Inner Space protocol, all these parts except the Data ACK could be located as Inner Options (the Checksum would not be necessary).

The MPTCP Data ACK has to remain as an Outer Option otherwise there would be a risk of flow control deadlock, as pointed out in [Raiciu12]. For instance, a Web client might pipeline multiple requests that fill a Web server’s receive buffer, while the Web server might be busy sending a large response to the first request before it reads the second request. If the Data ACK were an Inner Option, the Web client would have to stop acknowledging the first response from the server (due to lack of receive window). Then the server would not be able to move on to the next request--a classic deadlock.

The TCP-AO has to be located as an Outer Option to prevent the possibility of flow-control deadlock (because it would consume receive window on pure ACKs).

All sub-options of the tcpcrypt CRYPT option could be located as Inner Options. However, as long as the tcpcrypt MAC option covers

the TCP header and Outer Options, it has to be located as an Outer Option for the same deadlock reason as TCP-AO.

An Upgraded Server can support SYN Cookies [RFC4987] for Ordinary Connections. For Upgraded Connections Section 2.5 defines a new EchoCookie TCP option that is a prerequisite for InSpace implementations, and provides sufficient space for the more extensive connection state requirements of an InSpace server.

{ToDo: TCP States and Transitions, Connectionless Resets, ICMP Handling, Forward-Compatibility.}

4.2. Interaction with Middleboxes

The interaction with the assumptions about TCP made by middleboxes is covered extensively elsewhere:

- Section 2.3.3 specifies forwarding behaviour for Inner Options;
- The following sections explain the Inner Space protocol approach to middlebox traversal:
  - Section 3.2.1 justifies the magic number approach;
  - Section 3.2.2 explains why the protocol will remain robust as middlboxes evolve;
  - Section 3.2.4 justifies including Inner Options in sequence;
  - Section 3.2.5) explains how the protocol will remain robust to resegmentation.

4.3. Interaction with the Pre-Existing TCP API

An aim of the Inner Space protocol is for legacy applications to continue to just work without modification. Therefore it is expected that the dual handshaking logic and any maintenance of a cached white-list of servers that support the Inner Space protocol will be implemented beneath the well-known socket interface.

Inner Space implementations will need to comply with the following behaviours to ensure that legacy applications continue to receive predictable behaviour from the socket interface:

Querying local port (TCP client): If an application calls "getsockname()" while the TCP client behind the socket is engaged in a dual TCP handshake, the call SHOULD block until the local TCP

has aborted one of the connections so it knows which of the two ports will continue to be used.

Binding to an explicit port: If an application specifies that it wants the TCP client to use a specific port, the Inner Space capability MUST be disabled, because the dual handshake has to try two ports. Use of a specific port might be necessary, for example in a port-testing application or if the application wants to explicitly control all the handshaking logic of the Inner Space protocol itself.

Logging: The dual handshake will show up as a specific signature in logs of network activity. Log formats might not be able to record two local ports against one socket, so logs might contain unexpected or erroneous data. Even if logs correctly track both connection attempts, log analysis software might not expect to see one socket attempt to use two different ports. {ToDo: All this needs to be turned into a predictability requirement.}

Note that Inner Space has no impact on queries for the remote port from a TCP server. If an application calls "getpeername()" while the TCP server behind the socket is (unwittingly) engaged in a dual handshake, it will return the port of the remote client, even though this connection might subsequently be aborted. This is because a TCP server is not aware of whether it is part of a dual handshake.

It would be appropriate to enable the Inner Space protocol on a per-host or per-user basis. The necessary configuration switch does not need to be standardised, but it might allow the following three states:

Enabled: The stack will enable Inner Space on any TCP connection that that needs Inner Space for its TCP options. The stack might still disable the Inner Space protocol autonomously after the initial handshake if it is not needed.

Forwarding: The Forwarding mode is for TCP implementations on middleboxes that implement split TCP connections, as discussed in Section 2.3.3. Forwarding mode is similar to Disabled, except it forwards data in SYN without deferring it until the incoming connection is established.

Disabled: Inner Space is not enabled by default on any connections, except those that specifically request it.

The socket API might also need to be extended for future applications that want to control the Inner Space protocol explicitly. Experience

will determine the best API, so these ideas are merely informational suggestions at this stage:

Enabling/disabling Inner Space: As well as the above per-host or per-user switches, the extended API might need to allow an application to disable Inner Options on a per-socket basis (e.g. for testing). A socket might need to be opened in one of three possible Inner Space modes: i) Enabled; ii) Enabled initially but can be disabled autonomously by the stack if redundant; iii) Enabled initially, then disables itself after the SYN/ACK; and iv) Disabled. It also ought to be possible for an application to disable Inner Options on-demand mid-connection.

Querying support for Inner Space: An application might need to be able to determine whether the host supports Inner Space and in which mode it is enabled on a particular socket. For instance, an application might need to choose different socket options depending on whether Inner Space is enabled to make the necessary space available.

Latency vs Efficiency: A socket that prefers efficient use of connection state over latency might use the optional explicit variant of the dual handshake (Appendix B). It is unlikely that a new option specific to Inner Space would be needed to express this preference, as many operating systems already offer a similar socket option.

Logging: Log formats and log analysis software might need to be extended to distinguish between the deliberate RST within the dual handshake and an unexpected connection RST.

5. IANA Considerations

This specification requires IANA to allocate values from the TCP Option Kind name-space against the following names:

- "Inner Option Space Upgraded (InSpaceU)"
- "Inner Option Space Ordinary (InSpaceO)"
- "ModeSwitch"

Early implementation before the IANA allocation MUST follow [RFC6994] and use experimental option 254 and respective Experiment IDs:

- 0xUUUU (16 bits);
- 0xOOOO (16 bits);

o 0xMMMM (16 bits);

{ToDo: Values TBA and register them with IANA} then migrate to the assigned option after allocation.

6. Security Considerations

Certain cryptographic functions have different coverage rules for the TCP Header and TCP Payload. Placing some TCP options beyond the Data Offset could mean that they are treated differently from regular TCP options. This is a deliberate feature of the protocol, but application developers will need to be aware that this is the case.

A malicious host can send bogus SYN segments with a spoofed source IP address (a SYN flood attack). The Inner Space protocol does not alter the feasibility of this attack. However, the extra space for TCP options on a SYN allows the attacker to include more TCP options on a SYN than before, so it can make a server do more option processing before replying with a SYN/ACK. To mitigate this problme, a server under stress could deprioritise SYNs with longer option fields to focus its resources on SYNs that require less processing.

Each SYN in a SYN flood attack causes a TCP server to consume memory. The Inner Space protocol allows a potentially large amount of TCP option state to be negotiated during the SYN exchange, which could exhaust the TCP server’s memory. The EchoCookie TCP option (see Section 2.5) allows the server to place this state in a cookie and send it on the SYN/ACK to the purported address of the client--rather than hold it in memory.

Then, as long as the client returns the cookie on the acknowledgement and the server verifies it, the server can recover its full record of all the TCP options it negotiated and continue the connection without delay. On the other hand, the server’s responses to SYNs from spoofed addresses will scatter to those spoofed addresses and the server will not have consumed any memory while waiting in vain for them to reply. See the Security Considerations in [I-D.briscoe-tcpm-echo-cookie] for how the EchoCookie facility protects against reflection and amplification attacks.

7. Acknowledgements

The idea of this approach grew out of discussions with Joe Touch while developing draft-touch-tcpm-syn-ext-opt, and with Jana Iyengar and Olivier Bonaventure. The idea that it is architecturally preferable to place a protocol extension within a higher layer, and code its location into upgraded implementations of the lower layer, was originally articulated by Rob Hancock. {ToDo: Ref?} The following

people provided useful review comments: Joe Touch, Yuchung Cheng, John Leslie, Mirja Kuehlewind, Andrew Yourtchenko, Costin Raiciu, Marcelo Bagnulo Braun, Julian Chesterfield and Jaime Garcia.

Bob Briscoe's contribution is part-funded by the European Community under its Seventh Framework Programme through the Trilogy 2 project (ICT-317756) and the Reducing Internet Transport Latency (RITE) project (ICT-317700). The views expressed here are solely those of the author.

8. References

8.1. Normative References

[I-D.ietf-tcpm-fastopen]
Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", draft-ietf-tcpm-fastopen-10 (work in progress), September 2014.

[RFC0793] 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.

[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. ACM Internet Measurement Conference (IMC'11) 181--192, November 2011.

[I-D.bittau-tcpinc]
Bittau, A., Boneh, D., Hamburg, M., Handley, M., Mazieres, D., and Q. Slack, "Cryptographic protection of TCP Streams (tcpcrypt)", draft-bittau-tcpinc-01 (work in progress), July 2014.

[I-D.briscoe-tcpm-echo-cookie]
Briscoe, B., "The Echo Cookie TCP Option", draft-briscoe-tcpm-echo-cookie-00 (work in progress), October 2014.

[I-D.wing-tsvwg-happy-eyeballs-sctp]
Wing, D. and P. Natarajan, "Happy Eyeballs: Trending Towards Success with SCTP", draft-wing-tsvwg-happy-eyeballs-sctp-02 (work in progress), October 2010.

[Iyengar10]
Iyengar, J., Ford, B., Ailawadi, D., Amin, S., Nowlan, M., Tiwari, N., and J. Wise, "Minion--an All-Terrain Packet Packhorse to Jump-Start Stalled Internet Transports", Proc. Int'l Wkshp on Protocols for Future, Large-scale & Diverse Network Transports (PFLDnet'10) , November 2010.

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

[RFC2675] Borman, D., Deering, S., and R. Hinden, "IPv6 Jumbograms", RFC 2675, August 1999.

[RFC4987] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", RFC 4987, August 2007.

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

[RFC6555] Wing, D. and A. Yourtchenko, "Happy Eyeballs: Success with Dual-Stack Hosts", RFC 6555, April 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, "TCP Extensions for High Performance", RFC 7323, September 2014.

[Raiciu12]
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", Proc. USENIX Symposium on Networked Systems Design and Implementation , April 2012.

Appendix A. Protocol Extension Specifications

This appendix specifies protocol extensions that are OPTIONAL while the specification is experimental. If an implementation includes an extension, this section gives normative specification requirements.

However, if the extension is not implemented, the normative requirements can be ignored.

{Temporary note: The IETF may wish to consider making some of these extensions mandatory to implement if early testing shows they are useful or even necessary. Or it may wish to make at least the receiving side mandatory to implement to ensure that two-ended experiments are more feasible.}

A.1. Disabling InSpace and Generic Connection Mode Switching

This appendix is normative. It is separated from the body of the specification because it is OPTIONAL to implement while the Inner Space protocol is experimental. It defines the new ModeSwitch TCP option illustrated in Figure 5. This option provides a facility to disable the Inner Space protocol for the remainder of a connection. It also provides a general-purpose facility for a TCP connection to co-ordinate between the endpoints before switching into a yet-to-be-defined mode.

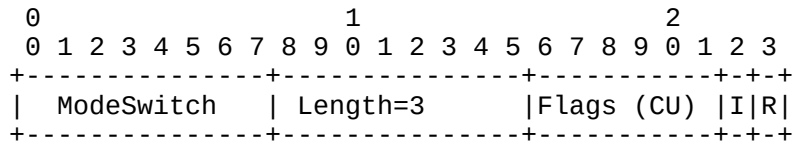


Figure 5: The ModeSwitch TCP Option

The Option Kind is ModeSwitch, the value of which is to be allocated by IANA {ToDo: Value TBA}. ModeSwitch MUST be used only as an Inner Option, because it uses the reliable ordered delivery property of Inner Options. Therefore implementation of the Inner Space protocol is REQUIRED for an implementation of ModeSwitch. Nonetheless, ModeSwitch is a generic facility for switching a connection between yet-to-be-defined modes that do not have to relate to extra option space.

The sender MUST set the option Length to 3 (octets). The Length field MUST be forwarded unchanged by other nodes, even if its value is different.

The Flags field is available for defining modes of the connection. Only two connection modes are currently defined. The first 6 bits of the Flags field are Currently Unused (CU) and the sender MUST set them to zero. The CU flags MUST be ignored and forwarded unchanged by other nodes, even if their value is non-zero.

The two 1-bit connection mode flags that are currently defined have the following meanings:

- R: Request flag if 1. Request mode is a special mode that allows the hosts to co-ordinate a change to any other mode(s);

- I: Inner Space mode: Enabled if 1, Disabled if 0.

The default Inner Space mode at the start of a connection is I=1, meaning Inner Space is in enabled mode.

The procedure for changing a mode or modes is as follows:

- The host that wants to change modes (the requester) sends a ModeSwitch message as an Inner Option with R=1 and with the other flag(s) set to the mode(s) it wants to change to. The requester does not change modes yet.

- The responder echoes the mode flag(s) it is willing to change to, with the request flag R=0.

- The half-connection from the responder changes to the mode(s) it confirms directly after the end of the segment that echoes its confirmation, i.e. after the last octet of the TCP Payload following the ModeSwitch option that echoes its confirmation. Therefore it sends the segment carrying the confirmation in the prior mode(s) of the connection.

- Once the requester receives the responder’s confirmation message, it re-echoes its confirmation of the responder’s confirmation, with the mode(s) set to those that both hosts agree on and R=0.

- The half-connection from the requester changes to the mode(s) it confirms directly after the end of the segment that re-echoes its confirmation. Therefore it sends the segment carrying the confirmation in the prior mode(s) of the connection.

- The responder can refuse a request to change into a mode in any one of three ways:

  * either implicitly by never confirming it;

  * or explicitly by sending a message with R=0 and the opposite mode;

  * or explicitly be sending a counter-request to switch to the opposite mode (that the connection is already in) with R=1.

The regular TCP sequence numbers and acknowledgement numbers of requests or confirmations can be used to disambiguate overlapping requests or responses.

Once a host switches to Disabled mode, it MUST NOT send any further InSpace Options. Therefore it can send no further Inner Options and it cannot switch back to Enabled mode for the rest of the connection.

To temporarily reduce InSpace overhead without permanently disabling the protocol, the sender can use a value of 0xFFFF in the Sent Payload Size (see Section 2.4).

A.2. Dual Handshake: The Explicit Variant

This appendix is normative. It is separated from the body of the specification because it is OPTIONAL to implement while the Inner Space protocol is experimental. It is not mandatory to implement because it will be more useful once the Inner Space protocol has become accepted widely enough that fewer middleboxes will discard SYN segments carrying this option (see Appendix B for when best to deploy it). It only works if both ends support it, but it can be deployed one end at a time, so there is no need for support in early experimental implementations.

{Temporary note: The choice between the explicit handshake in the present section or the handshake in Section 2.1.1 is a tradeoff between robustness against middlebox interference and minimal server state. During the IETF review process, one might be chosen as the only variant to go forward, at which point the other will be deleted. Alternatively, the IETF could require a server to understand both variants and a client could be implemented with either, or both. If both, the application could choose which to use at run-time. Then we will need a section describing the necessary API.}

This explicit dual handshake is similar to that in Section 2.1.1, except the SYN that the Upgraded Client sends on the Ordinary Connection is explicitly distinguishable from the SYN that would be sent by a Legacy Client. Then, if the server actually is an Upgraded Server, it can reset the Ordinary Connection itself, rather than creating connection state for at least a round trip until the client resets the connection.

For an explicit dual handshake, the TCP client still sends two alternative SYNs: a SYN-O intended for Legacy Servers and a SYN-U intended for Upgraded Servers. The two SYNs MUST have the same network addresses and the same destination port, but different source ports. Once the client establishes which type of server has responded, it continues the connection appropriate to that server

type and aborts the other. The SYN intended for Upgraded Servers includes additional options within the TCP Data (the SYN-U defined as before in Section 2.2.1).

Table 2 summarises the TCP 3-way handshake exchange for each of the two SYNs in the two right-hand columns, between an Upgraded TCP Client (the active opener) and either:

1. a Legacy Server, in the top half of the table (steps 2-4), or

2. an Upgraded Server, in the bottom half of the table (steps 2-4)

The table uses the same layout and symbols as Table 1, which has already been explained in Section 2.1.1.

		Ordinary Connection	Upgraded Connection
1	Upgraded Client	>SYN-O	>SYN-U
/\/\	/\/\/\/\/\/\/\	/\/\/\/\/\/\/\/\	/\/\/\/\/\/\/\/\
2	Legacy Server	<SYN/ACK	<SYN/ACK
3a	Upgraded Client	Waits for response to both SYNs	
3b	"	>ACK	>RST
4		Cont...	
/\/\	/\/\/\/\/\/\/\	/\/\/\/\/\/\/\/\	/\/\/\/\/\/\/\/\
2	Upgraded Server	<RST	<SYN/ACK-U
3	Upgraded Client		>ACK
4			Cont...

Table 2: Explicit Variant of Dual 3-Way Handshake in Two Server Scenarios

As before, an Upgraded Server MUST respond to a SYN-U with a SYN/ACK-U. Then, the client recognises that it is talking to an Upgraded Server.

Unlike before, an Upgraded Server MUST respond to a SYN-O with a RST. However, the client cannot rely on this behaviour, because a

middlebox might be stripping Outer TCP Options which would turn the SYN-O into a regular SYN before it reached the server. Then the handshake would effectively revert to the implicit variant. Therefore the client's behaviour still depends on which SYN-ACK arrives first, so its response to SYN-ACKs has to follow the rules specified for the implicit handshake variant in Section 2.1.1.

The rules for processing TCP options are also unchanged from those in Section 2.3.

A.2.1. SYN-O Structure

The SYN-O is merely a SYN with an extra InSpaceO Outer TCP Option as shown in Figure 6. It merely identifies that the SYN is opening an Ordinary Connection, but explicitly identifies that the client supports the Inner Space protocol.

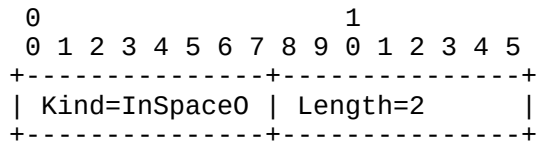


Figure 6: An InSpaceO TCP Option Flag

An InSpaceO TCP Option has Option Kind InSpaceO with value {ToDo: Value TBA} and MUST have Length = 2 octets.

To use this option, the client MUST place it with the Outer TCP Options. A Legacy Server will just ignore this TCP option, which is the normal behaviour for an option that TCP does not recognise [RFC0793].

A.2.2. Retransmission Behaviour - Explicit Variant

If the client receives a RST on one connection, but a short while after that {ToDo: duration TBA} the response to the SYN-U has not arrived, it SHOULD retransmit the SYN-U. If latency is more important than the extra TCP option space, in parallel to any retransmission, or instead of any retransmission, the client MAY send a SYN without any InSpace TCP Option, in case this is the cause of the black-hole. However, the presence of the RST implies that the SYN with the InSpaceO TCP Option (the SYN-O) probably reached the server, therefore it is more likely (but not certain) that the lack of response on the other connection is due to transmission loss or congestion loss.

If the client receives no response at all to either the SYN-O or the SYN-U, it SHOULD solely retransmit one or the other, not both. If latency is more important than the extra TCP option space, it SHOULD send a SYN without an InSpaceO TCP Option. Otherwise it SHOULD retransmit the SYN-U. It MUST NOT retransmit both segments, because the lack of response could be due to severe congestion.

A.2.3. Corner Cases

There is a small but finite possibility that the Explicit Dual Handshake might encounter the cases below. The Implicit Handshake (Section 2.1.1) is robust to these possibilities, but the Explicit Handshake is not, unless the following additional rules are followed:

Both successful: This could occur if one load-sharing replica of a server is upgraded, while another is not. This could happen in either order but, in both cases, the client aborts the last connection to respond:

* The client completes the Ordinary Handshake (because it receives a SYN/ACK), but then, before it has aborted the Upgraded Connection, it receives a SYN/ACK-U on it. In this case, the client MUST abort the Upgraded Connection even though it would work. Otherwise the client will have opened both connections, one with Inner TCP Options and one without. This could confuse the application.

* The client completes the Upgraded Connection after receiving a SYN/ACK-U, but then it receives a SYN/ACK in response to the SYN-O. In this case, the client MUST abort the connection it initiated with the SYN-O.

Both aborted: The client might receive a RST in response to its SYN-O, then an Ordinary SYN/ACK on its Upgraded Connection in response to its SYN-U. This could occur i) if a split connection middlebox actively forwards unknown options but holds back or discards data in a SYN; or ii) if one load-sharing replica of a server is upgraded, while another is not.

Whatever the likely cause, the client MUST still respond with a RST on its Upgraded Connection. Otherwise, its Inner TCP Options will be passed as user-data to the application by a Legacy Server.

If confronted with this scenario where both connections are aborted, the client will not be able to include extra options on a SYN, but it might still be able to set up a connection with extra option space on all the other segments in both directions using the approach in Appendix A.2.4. If that doesn’t work either, the

client's only recourse is to retry a new dual handshake on different source ports, or ultimately to fall-back to sending an Ordinary SYN.

A.2.4. Workround if Data in SYN is Blocked

If a path either holds back or discards data in a SYN-U, but there is evidence that the server is upgraded from a RST response to the SYN-O, the strategy below might at least allow a connection to use extra option space on all the segments except the SYN.

It is assumed that the symptoms described in the 'both aborted' case (Appendix A.2.3) have occurred, i.e. the server has responded to the SYN-O with a RST, but it has responded to the SYN-U with an Ordinary SYN/ACK not a SYN/ACK-U, so the client has had to RST the Upgraded Connection as well. In this case, the client SHOULD attempt the following (alternatively it MAY give up and fall back to opening an Ordinary TCP connection).

The client sends an 'Alternative SYN-U' by including an InSpaceU Outer TCP Option (Figure 7). This Alternative SYN-U merely flags that the client is attempting to open an Upgraded Connection. The client MUST NOT include any Inner Options or InSpace Option or Magic Number. If the previous aborted SYN/ACK-U acknowledged the data that the client sent within the original SYN-U, the client SHOULD resend the TCP Payload data in the Alternative SYN-U, otherwise it might as well defer it to the first data segment.

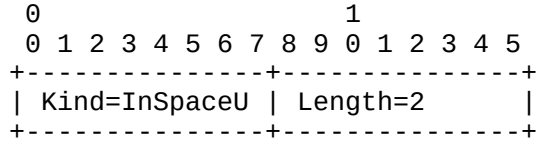


Figure 7: An InSpaceU Flag TCP option

An InSpaceU Flag TCP Option has Option Kind InSpaceU with value {ToDo: Value TBA} and MUST have Length = 2 octets.

To use this option, the client MUST place it with the Outer TCP Options. A Legacy Server will just ignore this TCP option, which is the normal behaviour for an option that TCP does not recognise [RFC0793]. Because the client has received a RST from the server in response to the SYN-O it can assume that the server is upgraded. So the client probably only needs to send a single Alternative SYN-U in this repeat attempt. Nonetheless, the RST might have been spurious.

Therefore the client MAY also send an Ordinary SYN in parallel, i.e. using the Implicit Dual Handshake (Section 2.1.1).

If an Upgraded Server receives a SYN carrying the InSpaceU option, it MUST continue the rest of the connection as if it had received a full SYN-U (Section 2.2), i.e. by processing any Outer Options in the SYN-U and responding with a SYN/ACK-U.

A.3. Jumbo InSpace TCP Option (only if SYN=0)

This appendix is normative. It is separated from the body of the specification because it is OPTIONAL to implement while the Inner Space protocol is experimental. In experimental implementations, it will be sufficient to implement the required behaviour for when the Length of a received InSpace Option is not recognised (Section 2.4).

If the IPv6 Jumbo extension header is used, the SentPayloadSize field will need to be 4 octets wide, not 2 octets. This section defines the format of the InSpace Option necessary to support jumbograms.

If sending a jumbogram, a sender MUST use the InSpace Option format defined in Figure 8. All the fields have the same meanings as defined in Section 2.2.2, except InOO and SentPayloadSize use more bits.

When reading a segment, the Jumbo InSpace Option could be present in a packet that is not a jumbogram (e.g. due to resegmentation). Therefore a receiver MUST use the Jumbo InSpace Option to work along the stream irrespective of whether arriving packets are jumbo sized or not.

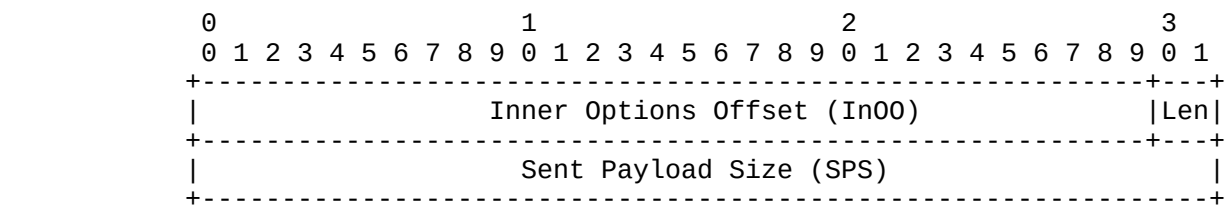


Figure 8: InSpace Option for a Jumbo Data-UNJH

A.4. Upgraded Segment Structure to Traverse DPI boxes

This appendix is normative. It is separated from the body of the specification because it is OPTIONAL to implement while the Inner Space protocol is experimental. If a receiver has implemented the Inner Space protocol but not this extension, no mechanism is provided for it to ask the sender to fall-back to the base Inner Space

protocol if it is sent a segment formatted according to this extension. However, it will at least fall-back naturally to regular TCP behaviour because of the dual handshake.

In experiments conducted between 2010 and 2011, [Honda11] reported that 7 of 142 paths (about 5%) blocked access to port 80 if the payload was not parsable as valid HTTP. This variant of the specification has been defined in case experiments prove that it significantly improves traversal of such deep packet inspection (DPI) boxes.

This variant starts the TCP Data with the expected app-layer headers on the first two segments in each direction:

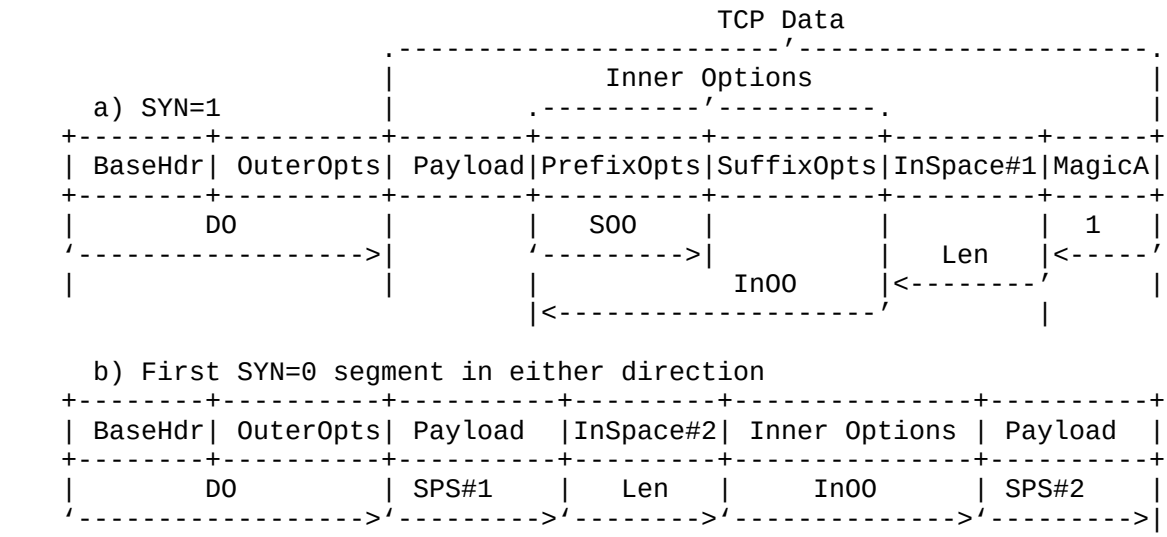
SYN=1: The structure in Figure 9a) is used on a SYN or SYN/ACK. The sender locates the 4-octet Magic Number A at the end of the segment. The sender right-aligns the 8-octet InSpace Option just before Magic Number A. Then it right-aligns the Inner Options against the InSpace Option, all after the end of the TCP Payload. The start of the Inner Options is therefore 4 * (InOO +3) octets before the end of the segment, where InOO is read from within the InSpace Option.

A receiver implementation will check whether Magic Number A is present at the end of the segment if it does not first find it at the start of the segment. Although the InnerOptions are located at the end of the TCP Payload, they are considered to be applied before the first octet of the TCP Payload.

SYN=0: The structure of the first non-SYN segment that contains any TCP Data is shown in Figure 9b).

The receiver will find the second InSpace Option (InSpace#2) located SPS#1 octets from the start of the segment, where SPS#1 is the value of Sent Payload Size that was read from the InSpace Option in the previous (SYN=1) segment that started the half-connection. Although the Inner Options are shifted, as for the first segment, they are still considered to be applied at the start of the TCP Data in this second segment.

From the second InSpace Option onwards, the structure of the stream reverts to that already defined in Section 2.2.1. So the value of Sent Payload Size (SPS#2) in the second InSpace Option (InSpace #2) defines the length of any remaining TCP Payload before the end of the first data segment, as shown.



All offsets are specified in 4-octet (32-bit) words, except SPS, which is in octets.

Figure 9: Segment Structures to Traverse DPI boxes (not to scale)

It is recognised that having to work from the end of the first segment makes processing more involved. Experimental implementation of this approach will determine whether the extra complexity improves DPI box traversal sufficiently to make it worthwhile.

Appendix B. Comparison of Alternatives

B.1. Implicit vs Explicit Dual Handshake

In the body of this specification, two variants of the dual handshake are defined:

1. The implicit dual handshake (Section 2.1.1) starting with just an Ordinary SYN (no InSpaceO flag option) on the Ordinary Connection;

2. The explicit dual handshake (Appendix A.2) starting with a SYN-O (InSpaceO flag option) on the Ordinary Connection.

Both schemes double up connection state (for a round trip) on the Legacy Server. But only the implicit scheme doubles up connection state (for a round trip) on the Upgraded Server as well. On the other hand, the explicit scheme risks delay accessing a Legacy Server

if a middlebox discards the SYN-O (it is possible that some firewalls will discard packets with unrecognised TCP options {ToDo: ref?}). Table 3 summarises these points.

	SYN (Implicit)	SYN-L (Explicit)
Minimum state on Upgraded Server	-	+
Minimum risk of delay to Legacy Server	+	-

Table 3: Comparison of Implicit vs. Explicit Dual Handshake on the Ordinary Connection

There is no need for the IETF to choose between these. If the specification allows either or both, the tradeoff can be left to implementers at build-time, or to the application at run-time.

Initially clients might choose the Implicit Dual Handshake to minimise delays due to middlebox interference. But later, perhaps once more middleboxes support the scheme, clients might choose the Explicit scheme, to minimise state on Upgraded Servers.

Appendix C. Protocol Design Issues (to be Deleted before Publication)

This appendix is informative, not normative. It records outstanding issues with the protocol design that will need to be resolved before publication.

Option alignment following re-segmentation: If the byte-stream is resegmented (e.g. by a connection splitter), the TCP options within the stream will not necessarily align on 4-octet word boundaries within the new segments.

Ossifies reliable ordered delivery into TCP design: At present it is theoretically possible to implement a variant of TCP that provides partial reliability. Inner Space as it stands would prevent a future partial reliable TCP, but not if out-of-order delivery were added, as discussed below.

Ideally Outer Options in Inner: Ideally enable Outer Options to be located beyond the Data Offset: i) without consuming receive window ii) either without consuming sequence space or, if otherwise, must be robust to middlebox correction; iii) delivered immediately on reception, not in sent order. Could use the Minion

[Iyengar10] variant (or a similar variant) of the consistent overhead byte-stuffing (COBS) encoding.

Appendix D. Change Log (to be Deleted before Publication)

A detailed version history can be accessed at <http://datatracker.ietf.org/doc/draft-briscoe-tcpm-inner-space/history/>

From briscoe-...-inner-space-00 to briscoe-...-inner-space-01:

Technical changes:

* Corrected DO to 4 * DO (twice)
* Confirmed that receive window applies to Inner Options
* Generalised the cause of decryption/decompression from a previous TCP option to any previouis control message
* Added requirement for a middlebox not to defer data on SYN
* Latency of dual handshake is worst of two
* Completed "Interaction with Pre-Existing TCP Implementations" section, covering other TCP variants, TCP in middleboxes and the TCP API. Shifted some TCP options to Outer only, because of RWND deadlock problem
* Added two outstanding issues: i) ossifies reliable ordered delivery; ii) Ideally Outer in Inner.

Editorial changes:

* Removed section on Echo TCP option to a separate I-D that is mandatory to implement for inner-space, and shifted some SYN flood discussion in Security Considerations
* Clarifications throughout
* Acknowledged more review comments

From draft-briscoe-tcpm-syn-op-sis-02 to draft-briscoe-tcpm-inner-space-00:

The Inner Space protocol is a development of a proposal called the SynOpSis (Sister SYN options) protocol. Most of the elements of Inner Space were in SynOpSis, such as the implicit and explicit dual handshakes; the use of a magic number to flag the existence

of the option; the various header offsets; and the option processing rules.

The main technical differences are: Inner Space extends option space on any segment, not just the SYN; this advance requires the introduction of the Sent Payload Size field and a general rearrangement and simplification of the protocol format; the option processing rules have been extended to assure compatibility with TFO and one degree of recursion has been introduced to cater for encryption or compression of Inner Options; The Echo option has been added to provide a SYN-cookie-like capability. Also, the default protocol has been pared down to the bare bones and optional extensions relegated to appendices.

The main editorial differences are: The emphasis of the Abstract and Introduction has expanded from a focus on just extra space using the dual handshake to include much more comprehensive middlebox traversal. A comprehensive Design Rationale section has been added.

Author’s Address

Bob Briscoe
BT
B54/77, Adastral Park
Martlesham Heath
Ipswich IP5 3RE
UK

Phone: +44 1473 645196
Email: bob.briscoe@bt.com
URI: http://bobbriscoe.net/

TCPM Working Group
Internet Draft

J. Heitz
Cisco
Chuan He
Ericsson

Intended status: Informational
Expires: April 2015

October 19, 2014

TCP Retransmission Timer for Virtual Machines
draft-heitzhe-tcpm-vm-rto-00.txt

Abstract

A Round Trip Time (RTT) estimate that decays performs badly in a bursty environment. A round trip time estimator that does not decay for a period of time is proposed.

It does not require a minimum value to be configured. It works equally well when the typical RTT is 100uS as when it is 10 seconds.

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 March 19, 2009.

Table of Contents

1. Introduction

Virtual machines can create bursty environments for TCP, especially if the TCP also runs in a thread within a process of a busy host machine. Round trip time measurements can frequently be hundreds of times the average. A retransmission timer (RTO) calculation that decays even a little bit for each small measured RTT will cause a retransmission for every such outlier RTT.

[RFC6298] requires a minimum RTO of 1 second to avoid spurious retransmissions. RTTs between VMs in a lightly loaded host are regularly less than 1 millisecond. On a heavily loaded host, RTTs do not all get higher. They get a little higher in the median, to a few milliseconds, but the number of spikes of 100s of milliseconds increases. The EWMA algorithm of [RFC6298] is hardly used. It simply defaults to 1 second. When the default is reduced, it gets a spurious retransmission on nearly every spike.

The proposed algorithm is less aggressive than that of [RFC6298] and needs no minimum setting. In fact, it grew out of an attempt to find a better minimum.

The retransmission timer is a compromise. If it is set too low, then spurious retransmissions occur, but if it is set too high, then it takes too long to retransmit when it is really needed.

The right balance is achieved when an acceptably small number of spurious retransmissions occur.

2. The Algorithm

The basis of the algorithm is as follows: Time is divided into intervals. Within each interval, the highest RTT is determined. This RTT forms the RTO to be used for the next interval. The RTO is constant for the duration of an interval.

The end of an interval and the beginning of the next one is determined when any of the following events occur:

1. An RTT is measured that is greater than the maximum RTT from the previous interval. The maximum RTT from the previous interval is the RTT that determines the RTO of the current interval. This measured RTT is the greatest RTT measured for the current interval. It is considered part of the current interval, not of the next one.

2. A large number (suggest 20) of windows of data has been transmitted.

The RTO of one interval is the maximum RTT of the previous interval plus some headroom. The suggested headroom is 1/4, so RTO = 1.25 * (previous max RTT).

A window of data is the largest ever advertised window of a session.

2.1. Initial Interval

The RTO of the first interval should be 1 second. The length of the first interval should be shorter than the others. Suggestion is 3 RTT measurements. The initial RTO may alternatively be determined from a history of previous connections.

An alternative is to run the regular algorithm as in [RFC6298] during the first interval. The RTTs would still be individually measured in preparation for the second interval.

3. Delayed ACK Timer

The delayed ACK timer is not handled differently. If a delayed ACK timer is in effect on the peer, it may cause high RTT measurements. If delayed ACK happens less than every 20 windows, then it will be included as part of the maximum RTT measurement. If it happens after more than 20 windows of data have been transmitted, then a possibly resulting retransmission is not excessive.

4. Backing off the timer

This document is an alternative to section 2 of [RFC6298]. In particular, the backing off mechanism in section 5 remains intact.

5. Security Considerations

No security issues beyond those outlined in [RFC6298] have been identified.

6. IANA Considerations

None

7. References

7.1. Normative References

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

8. Acknowledgments

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

Authors' Addresses

Jakob Heitz
Cisco
510 McCarthy Blvd,
Milpitas, CA 95035

Email: jheitz@cisco.com

Chuan He
Ericsson
300 Holger Way,
San Jose, CA 95134

Email: chuan.he@ericsson.com

TCP Maintenance and Minor Extensions (tcpm)
Internet-Draft
Intended status: Experimental
Expires: May 8, 2016

P. Hurtig
A. Brunstrom
Karlstad University
A. Petlund
Simula Research Laboratory AS
M. Welzl
University of Oslo

November 5, 2015

TCP and SCTP RTO Restart
draft-ietf-tcpm-rtorestart-10

Abstract

This document describes a modified sender-side algorithm for managing the TCP and SCTP retransmission timers that provides faster loss recovery when there is a small amount of outstanding data for a connection. The modification, RTO Restart (RTOR), allows the transport to restart its retransmission timer using a smaller timeout duration, so that the effective RTO becomes more aggressive in situations where fast retransmit cannot be used. This enables faster loss detection and recovery for connections that are short-lived or application-limited.

Status of This Memo

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

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

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

This Internet-Draft will expire on May 8, 2016.

Copyright Notice

1. Introduction

TCP and SCTP use two almost identical mechanisms to detect and recover from data loss, specified in [RFC6298][RFC5681] (for TCP) and [RFC4960] (for SCTP). First, if transmitted data is not acknowledged within a certain amount of time, a retransmission timeout (RTO) occurs, and the data is retransmitted. While the RTO is based on measured round-trip times (RTTs) between the sender and receiver, it also has a conservative lower bound of 1 second to ensure that delayed data are not mistaken as lost. Second, when a sender receives duplicate acknowledgments, or similar information via selective acknowledgments, the fast retransmit algorithm suspects data loss and can trigger a retransmission. Duplicate (and selective) acknowledgments are generated by a receiver when data arrives out-of-order. As both data loss and data reordering cause out-of-order arrival, fast retransmit waits for three out-of-order notifications before considering the corresponding data as lost. In some situations, however, the amount of outstanding data is not enough to trigger three such acknowledgments, and the sender must rely on lengthy RTOs for loss recovery.

The amount of outstanding data can be small for several reasons:

(1) The connection is limited by the congestion control when the path has a low total capacity (bandwidth-delay product) or the connection's share of the capacity is small. It is also limited by the congestion control in the first few RTTs of a connection or after an RTO when the available capacity is probed using slow-start.

(2) The connection is limited by the receiver's available buffer space.

(3) The connection is limited by the application if the available capacity of the path is not fully utilized (e.g. interactive applications), or at the end of a transfer.

While the reasons listed above are valid for any flow, the third reason is most common for applications that transmit short flows, or use a bursty transmission pattern. A typical example of applications that produce short flows are web-based applications. [RJ10] shows that 70% of all web objects, found at the top 500 sites, are too small for fast retransmit to work. [FDT13] shows that about 77% of all retransmissions sent by a major web service are sent after RTO expiry. Applications with bursty transmission patterns often send data in response to actions, or as a reaction to real life events. Typical examples of such applications are stock trading systems, remote computer operations, online games, and web-based applications using persistent connections. What is special about this class of applications is that they often are time-dependant, and extra latency can reduce the application service level [P09].

The RTO Restart (RTOR) mechanism described in this document makes the effective RTO slightly more aggressive when the amount of outstanding data is too small for fast retransmit to work, in an attempt to enable faster loss recovery while being robust to reordering. While RTOR still conforms to the requirement for when a segment can be retransmitted, specified in [RFC6298] (for TCP) and [RFC4960] (for SCTP) it could increase the risk of spurious timeouts. To determine whether this modification is safe to deploy and enable by default, further experimentation is required. Section 5 discusses experiments still needed, including evaluations in environments where the risk of spurious retransmissions are increased e.g. mobile networks with highly varying RTTs.

The remainder of this document describes RTOR and its implementation for TCP only, to make the document easier to read. However, the RTOR algorithm described in Section 4 is applicable also for SCTP. Furthermore, Section 7 details the SCTP socket API needed to control RTOR.

2. Terminology

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

This document introduces the following variables:

The number of previously unsent segments (prevunsnt): The number of segments that a sender has queued for transmission, but has not yet sent.

RTO Restart threshold (rrthresh): RTOR is enabled whenever the sum of the number of outstanding and previously unsent segments (prevunsnt) is below this threshold.

3. RTO Overview and Rationale for RTOR

The RTO management algorithm described in [RFC6298] recommends that the retransmission timer is restarted when an acknowledgment (ACK) that acknowledges new data is received and there is still outstanding data. The restart is conducted to guarantee that unacknowledged segments will be retransmitted after approximately RTO seconds. The standardized RTO timer management is illustrated in Figure 1 where a TCP sender transmits three segments to a receiver. The arrival of the first and second segment triggers a delayed ACK (delACK) [RFC1122], which restarts the RTO timer at the sender. The RTO is restarted approximately one RTT after the transmission of the third segment. Thus, if the third segment is lost, as indicated in Figure 1, the effective loss detection time become "RTO + RTT" seconds. In some situations, the effective loss detection time becomes even longer. Consider a scenario where only two segments are outstanding. If the second segment is lost, the time to expire the delACK timer will also be included in the effective loss detection time.

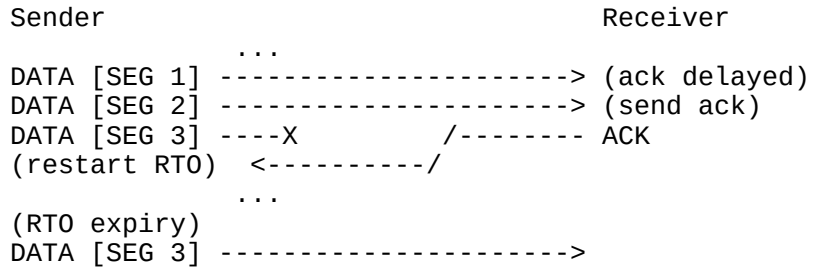


Figure 1: RTO restart example

For bulk traffic the current approach is beneficial -- it is described in [EL04] to act as a "safety margin" that compensates for some of the problems that the authors have identified with the standard RTO calculation. Notably, the authors of [EL04] also state that "this safety margin does not exist for highly interactive applications where often only a single packet is in flight". In general, however, as long as enough segments arrive at a receiver to enable fast retransmit, RTO-based loss recovery should be avoided. RTOs should only be used as a last resort, as they drastically lower the congestion window compared to fast retransmit.

Although fast retransmit is preferrable there are situations where timeouts are appropriate, or the only choice. For example, if the network is severely congested and no segments arrive RTO-based recovery should be used. In this situation, the time to recover from the loss(es) will not be the performance bottleneck. However, for connections that do not utilize enough capacity to enable fast retransmit, RTO-based loss detection is the only choice and the time required for this can become a performance bottleneck.

4. RTOR Algorithm

To enable faster loss recovery for connections that are unable to use fast retransmit, RTOR can be used. This section specifies the modifications required to use RTOR. By resetting the timer to "RTO - T_earliest", where T_earliest is the time elapsed since the earliest outstanding segment was transmitted, retransmissions will always occur after exactly RTO seconds.

This document specifies an OPTIONAL sender-only modification to TCP and SCTP which updates step 5.3 in Section 5 of [RFC6298] (and a similar update in Section 6.3.2 of [RFC4960] for SCTP). A sender that implements this method MUST follow the algorithm below:

When an ACK is received that acknowledges new data:

(1) Set T_earliest = 0.

(2) If the sum of the number of outstanding and previously unsent segments (prevunsnt) is less than an RTOR threshold (rrthresh), set T_earliest to the time elapsed since the earliest outstanding segment was sent.

(3) Restart the retransmission timer so that it will expire after (for the current value of RTO):

(a) RTO - T_earliest, if RTO - T_earliest > 0.

(b) RTO, otherwise.

The RECOMMENDED value of rrthresh is four, as this value will ensure that RTOR is only used when fast retransmit cannot be triggered. With this update, TCP implementations MUST track the time elapsed since the transmission of the earliest outstanding segment (T_earliest). As RTOR is only used when the amount of outstanding and previously unsent data is less than rrthresh segments, TCP implementations also need to track whether the amount of outstanding and previously unsent data is more, equal, or less than rrthresh segments. Although some packet-based TCP implementations (e.g.

Linux TCP) already track both the transmission times of all segments and also the number of outstanding segments, not all implementations do. Section 5.3 describes how to implement segment tracking for a general TCP implementation. To use RTOR, the calculated expiration time MUST be positive (step 3(a) in the list above); this is required to ensure that RTOR does not trigger retransmissions prematurely when previously retransmitted segments are acknowledged.

5. Discussion

Although RTOR conforms to the requirement in [RFC6298] that segments must not be retransmitted earlier than RTO seconds after their original transmission, RTOR makes the effective RTO more aggressive. In this section, we discuss the applicability and the issues related to RTOR.

5.1. Applicability

The currently standardized algorithm has been shown to add at least one RTT to the loss recovery process in TCP [LS00] and SCTP [HB11][PBP09]. For applications that have strict timing requirements (e.g. interactive web) rather than throughput requirements, using RTOR could be beneficial because the RTT and also the delACK timer of receivers are often large components of the effective loss recovery time. Measurements in [HB11] have shown that the total transfer time of a lost segment (including the original transmission time and the loss recovery time) can be reduced by 35% using RTOR. These results match those presented in [PGH06][PBP09], where RTOR is shown to significantly reduce retransmission latency.

There are also traffic types that do not benefit from RTOR. One example of such traffic is bulk transmission. The reason why bulk traffic does not benefit from RTOR is that such traffic flows mostly have four or more segments outstanding, allowing loss recovery by fast retransmit. However, there is no harm in using RTOR for such traffic as the algorithm only is active when the amount of outstanding and unsent segments are less than rrthresh (default 4).

Given that RTOR is a mostly conservative algorithm, it is suitable for experimentation as a system-wide default for TCP traffic.

5.2. Spurious Timeouts

RTOR can in some situations reduce the loss detection time and thereby increase the risk of spurious timeouts. In theory, the retransmission timer has a lower bound of 1 second [RFC6298], which limits the risk of having spurious timeouts. However, in practice most implementations use a significantly lower value. Initial

measurements show slight increases in the number of spurious timeouts when such lower values are used [RHB15]. However, further experiments, in different environments and with different types of traffic, are encouraged to quantify such increases more reliably.

Does a slightly increased risk matter? Generally, spurious timeouts have a negative effect on the network as segments are transmitted needlessly. However, recent experiments do not show a significant increase in network load for a number of realistic scenarios [RHB15]. Another problem with spurious retransmissions is related to the performance of TCP/SCTP, as the congestion window is reduced to one segment when timeouts occur [RFC5681]. This could be a potential problem for applications transmitting multiple bursts of data within a single flow, e.g. web-based HTTP/1.1 and HTTP/2.0 applications. However, results from recent experiments involving persistent web traffic [RHB15] revealed a net gain of using RTOR. Other types of flows, e.g. long-lived bulk flows, are not affected as the algorithm is only applied when the amount of outstanding and unsent segments is less than rrthresh. Furthermore, short-lived and application-limited flows are typically not affected as they are too short to experience the effect of congestion control or have a transmission rate that is quickly attainable.

While a slight increase in spurious timeouts has been observed using RTOR, it is not clear whether the effects of this increase mandate any future algorithmic changes or not -- especially since most modern operating systems already include mechanisms to detect [RFC3522][RFC3708][RFC5682] and resolve [RFC4015] possible problems with spurious retransmissions. Further experimentation is needed to determine this and thereby move this specification from experimental to the standards track. For instance, RTOR has not been evaluated in the context of mobile networks. Mobile networks often incur highly variable RTTs (delay spikes), due to e.g. handovers, and would therefore be a useful scenario for further experimentation.

5.3. Tracking Outstanding and Previously Unsent Segments

The method of tracking outstanding and previously unsent segments will probably differ depending on the actual TCP implementation. For packet-based TCP implementations, tracking outstanding segments is often straightforward and can be implemented using a simple counter. For byte-based TCP stacks it is a more complex task. Section 3.2 of [RFC5827] outlines a general method of tracking the number of outstanding segments. The same method can be used for RTOR. The implementation will have to track segment boundaries to form an understanding as to how many actual segments have been transmitted, but not acknowledged. This can be done by the sender tracking the boundaries of the rrthresh segments on the right side of the current

window (which involves tracking rrthresh + 1 sequence numbers in TCP). This could be done by keeping a circular list of the segment boundaries, for instance. Cumulative ACKs that do not fall within this region indicate that at least rrthresh segments are outstanding, and therefore RTOR is not enabled. When the outstanding window becomes small enough that RTOR can be invoked, a full understanding of the number of outstanding segments will be available from the rrthresh + 1 sequence numbers retained. (Note: the implicit sequence number consumed by the TCP FIN bit can also be included in the tracking of segment boundaries.)

Tracking the number of previously unsent segments depends on the segmentation strategy used by the TCP implementation, not whether it is packet-based or byte-based. In the case segments are formed directly on socket writes, the process of determining the number of previously unsent segments should be trivial. In the case that unsent data can be segmented (or re-segmented) as long as it still is unsent, a straightforward strategy could be to divide the amount of unsent data (in bytes) with the SMSS to obtain an estimate. In some cases, such an estimation could be too simplistic, depending on the segmentation strategy of the TCP implementation. However, this estimation is not critical to RTOR. The tracking of prevunsnt is only made to optimize a corner case in which RTOR was unnecessarily disabled. Implementations can use a simplified method by setting prevunsnt to rrthresh whenever previously unsent data is available, and set prevunsnt to zero when no new data is available. This will disable RTOR in the presence of unsent data and only use the number of outstanding segments to enable/disable RTOR.

6. Related Work

There are several proposals that address the problem of not having enough ACKs for loss recovery. In what follows, we explain why the mechanism described here is complementary to these approaches:

The limited transmit mechanism [RFC3042] allows a TCP sender to transmit a previously unsent segment for each of the first two dupACKs. By transmitting new segments, the sender attempts to generate additional dupACKs to enable fast retransmit. However, limited transmit does not help if no previously unsent data is ready for transmission. [RFC5827] specifies an early retransmit algorithm to enable fast loss recovery in such situations. By dynamically lowering the number of dupACKs needed for fast retransmit (dupthresh), based on the number of outstanding segments, a smaller number of dupACKs is needed to trigger a retransmission. In some situations, however, the algorithm is of no use or might not work properly. First, if a single segment is outstanding, and lost, it is impossible to use early retransmit. Second, if ACKs are lost, early

retransmit cannot help. Third, if the network path reorders segments, the algorithm might cause more spurious retransmissions than fast retransmit. The recommended value of RTOR's rrthresh variable is based on the dupthresh, but is possible to adapt to allow tighter integration with other experimental algorithms such as early retransmit.

Tail Loss Probe [TLP] is a proposal to send up to two "probe segments" when a timer fires which is set to a value smaller than the RTO. A "probe segment" is a new segment if new data is available, else a retransmission. The intention is to compensate for sluggish RTO behavior in situations where the RTO greatly exceeds the RTT, which, according to measurements reported in [TLP], is not uncommon. Furthermore, TLP also tries to circumvent the congestion window reset to one segment by instead enabling fast recovery. The Probe timeout (PTO) is normally two RTTs, and a spurious PTO is less risky than a spurious RTO because it would not have the same negative effects (clearing the scoreboard and restarting with slow-start). TLP is a more advanced mechanism than RTOR, requiring e.g. SACK to work, and is often able to reduce loss recovery times more. However, it also increases the amount of spurious retransmissions noticeably, as compared to RTOR [RHB15].

TLP is applicable in situations where RTOR does not apply, and it could overrule (yielding a similar general behavior, but with a lower timeout) RTOR in cases where the number of outstanding segments is smaller than four and no new segments are available for transmission. The PTO has the same inherent problem of restarting the timer on an incoming ACK, and could be combined with a strategy similar to RTOR's to offer more consistent timeouts.

7. SCTP Socket API Considerations

This section describes how the socket API for SCTP defined in [RFC6458] is extended to control the usage of RTO restart for SCTP.

Please note that this section is informational only.

7.1. Data Types

This section uses data types from [IEEE.1003-1G.1997]: uintN_t means an unsigned integer of exactly N bits (e.g., uint16_t). This is the same as in [RFC6458].

7.2. Socket Option for Controlling the RTO Restart Support (SCTP_RTO_RESTART)

This socket option allows the enabling or disabling of RTO Restart for SCTP associations.

Whether RTO Restart is enabled or not per default is implementation specific.

This socket option uses IPPROTO_SCTP as its level and SCTP_RTO_RESTART as its name. It can be used with getsockopt() and setsockopt(). The socket option value uses the following structure defined in [RFC6458]:

```
struct sctp_assoc_value {
  sctp_assoc_t assoc_id;
  uint32_t assoc_value;
};
```

assoc_id: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets, this parameter indicates upon which association the user is performing an action. The special sctp_assoc_t SCTP_{FUTURE|CURRENT|ALL}_ASSOC can also be used in assoc_id for setsockopt(). For getsockopt(), the special value SCTP_FUTURE_ASSOC can be used in assoc_id, but it is an error to use SCTP_{CURRENT|ALL}_ASSOC in assoc_id.

assoc_value: A non-zero value encodes the enabling of RTO restart whereas a value of 0 encodes the disabling of RTO restart.

sctp_opt_info() needs to be extended to support SCTP_RTO_RESTART.

8. IANA Considerations

This memo includes no request to IANA.

9. Security Considerations

This document specifies an experimental sender-only modification to TCP and SCTP. The modification introduces a change in how to set the retransmission timer’s value when restarted. Therefore, the security considerations found in [RFC6298] apply to this document. No additional security problems have been identified with RTO Restart at this time.

10. Acknowledgements

The authors wish to thank Michael Tuexen for contributing the SCTP Socket API considerations and Godred Fairhurst, Yuchung Cheng, Mark Allman, Anantha Ramaiah, Richard Scheffenegger, Nicolas Kuhn, Alexander Zimmermann, and Michael Scharf for commenting on the draft and the ideas behind it.

All the authors are supported by RITE (http://riteproject.eu/), a research project (ICT-317700) funded by the European Community under its Seventh Framework Program. The views expressed here are those of the author(s) only. The European Commission is not liable for any use that may be made of the information in this document.

11. Changes from Previous Versions

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

11.1. Changed from draft-ietf-...-09 to -10

- Changed wording in abstract, from "delay" to "timeout duration".

11.2. Changed from draft-ietf-...-08 to -09

- Clarified, in the abstract, that the modified restart causes a smaller retransmission delay in total.
- Clarified, in the introduction, that the fast retransmit algorithm may cause retransmissions upon receiving duplicate acknowledgments, not that it unconditionally does so.
- Changed wording from "to proposed standard" to "to the standards track".
- Changed algorithm description so that a TCP sender MUST track the time elapsed since the transmission of the earliest outstanding segment. This was not explicitly stated in previous versions of the draft.

11.3. Changes from draft-ietf-...-07 to -08

- Clarified, at multiple places in the document, that the modification only causes the effective RTO to be more aggressive, not the actual RTO.
- Removed information in the introduction that was too detailed, i.e., material that is hard to understand without knowing details of the algorithm.

- Changed the name of Section 3 to more correctly capture the actual contents of the section.
- Re-arranged the text in Section 3 to have a more logical structure.
- Moved text from the algorithm description (Section 4) to the introduction of the discussion section (Section 5). The text was discussing the possible effects of the algorithm more than describing the actual algorithm.
- Clarified why the RECOMMENDED value of rrthresh is four.
- Reworked the introduction to be suitable for both TCP and SCTP.

11.4. Changes from draft-ietf-...-06 to -07

- Clarified, at multiple places in the document, that the modification is sender-only.
- Added an explanation (in the introduction) to why the mechanism is experimental and what experiments are missing.
- Added a sentence in Section 4 to clarify that the section is the one describing the actual modification.

11.5. Changes from draft-ietf-...-05 to -06

- Added socket API considerations, after discussing the draft in tsvwg.

11.6. Changes from draft-ietf-...-04 to -05

- Introduced variable to track the number of previously unsent segments.
- Clarified many concepts, e.g. extended the description on how to track outstanding and previously unsent segments.
- Added a reference to initial measurements on the effects of using RTOR.
- Improved wording throughout the document.

11.7. Changes from draft-ietf-...-03 to -04

- Changed the algorithm to allow RTOR when there is unsent data available, but the cwnd does not allow transmission.
- Changed the algorithm to not trigger if RTOR <= 0.
- Made minor adjustments throughout the document to adjust for the algorithmic change.
- Improved the wording throughout the document.

11.8. Changes from draft-ietf-...-02 to -03

- Updated the document to use "RTOR" instead of "RTO Restart" when refering to the modified algorithm.
- Moved document terminology to a section of its own.
- Introduced the rrthresh variable in the terminology section.
- Added a section to generalize the tracking of outstanding segments.
- Updated the algorithm to work when the number of outstanding segments is less than four and one segment is ready for transmission, by restarting the timer when new data has been sent.
- Clarified the relationship between fast retransmit and RTOR.
- Improved the wording throughout the document.

11.9. Changes from draft-ietf-...-01 to -02

- Changed the algorithm description in Section 3 to use formal RFC 2119 language.
- Changed last paragraph of Section 3 to clarify why the RTO restart algorithm is active when less than four segments are outstanding.
- Added two paragraphs in Section 4.1 to clarify why the algorithm can be turned on for all TCP traffic without having any negative effects on traffic patterns that do not benefit from a modified timer restart.
- Improved the wording throughout the document.
- Replaced and updated some references.

11.10. Changes from draft-ietf-...-00 to -01

- Improved the wording throughout the document.
- Removed the possibility for a connection limited by the receiver's advertised window to use RTO restart, decreasing the risk of spurious retransmission timeouts.
- Added a section that discusses the applicability of and problems related to the RTO restart mechanism.
- Updated the text describing the relationship to TLP to reflect updates made in this draft.
- Added acknowledgments.

12. References

12.1. Normative References

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

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

[RFC3042] Allman, M., Balakrishnan, H., and S. Floyd, "Enhancing TCP's Loss Recovery Using Limited Transmit", RFC 3042, DOI 10.17487/RFC3042, January 2001, <http://www.rfc-editor.org/info/rfc3042>.

[RFC3522] Ludwig, R. and M. Meyer, "The Eifel Detection Algorithm for TCP", RFC 3522, DOI 10.17487/RFC3522, April 2003, <http://www.rfc-editor.org/info/rfc3522>.

[RFC3708] Blanton, E. and M. Allman, "Using TCP Duplicate Selective Acknowledgement (DSACKs) and Stream Control Transmission Protocol (SCTP) Duplicate Transmission Sequence Numbers (TSNs) to Detect Spurious Retransmissions", RFC 3708, DOI 10.17487/RFC3708, February 2004, <http://www.rfc-editor.org/info/rfc3708>.

[RFC4015] Ludwig, R. and A. Gurtov, "The Eifel Response Algorithm for TCP", RFC 4015, DOI 10.17487/RFC4015, February 2005, <http://www.rfc-editor.org/info/rfc4015>.

[RFC4960] Stewart, R., Ed., "Stream Control Transmission Protocol", RFC 4960, DOI 10.17487/RFC4960, September 2007, <http://www.rfc-editor.org/info/rfc4960>.

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

[RFC5682] Sarolahti, P., Kojo, M., Yamamoto, K., and M. Hata, "Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP", RFC 5682, DOI 10.17487/RFC5682, September 2009, <http://www.rfc-editor.org/info/rfc5682>.

[RFC5827] Allman, M., Avrachenkov, K., Ayesta, U., Blanton, J., and P. Hurtig, "Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)", RFC 5827, DOI 10.17487/RFC5827, May 2010, <http://www.rfc-editor.org/info/rfc5827>.

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

12.2. Informative References

[EL04] Ekstroem, H. and R. Ludwig, "The Peak-Hopper: A New End-to-End Retransmission Timer for Reliable Unicast Transport", IEEE INFOCOM 2004, March 2004.

[FDT13] Flach, T., Dukkipati, N., Terzis, A., Raghavan, B., Cardwell, N., Cheng, Y., Jain, A., Hao, S., Katz-Bassett, E., and R. Govindan, "Reducing Web Latency: the Virtue of Gentle Aggression", Proc. ACM SIGCOMM Conf., August 2013.

[HB11] Hurtig, P. and A. Brunstrom, "SCTP: designed for timely message delivery?", Springer Telecommunication Systems 47 (3-4), August 2011.

[IEEE.1003-1G.1997] Institute of Electrical and Electronics Engineers, "Protocol Independent Interfaces", IEEE Standard 1003.1G, March 1997.

[LS00] Ludwig, R. and K. Sklower, "The Eifel retransmission timer", ACM SIGCOMM Comput. Commun. Rev., 30(3), July 2000.

[P09] Petlund, A., "Improving latency for interactive, thin-stream applications over reliable transport", Unipub PhD Thesis, Oct 2009.

[PBP09] Petlund, A., Beskow, P., Pedersen, J., Paaby, E., Griwodz, C., and P. Halvorsen, "Improving SCTP Retransmission Delays for Time-Dependent Thin Streams", Springer Multimedia Tools and Applications, 45(1-3), 2009.

[PGH06] Pedersen, J., Griwodz, C., and P. Halvorsen, "Considerations of SCTP Retransmission Delays for Thin Streams", IEEE LCN 2006, November 2006.

[RFC6458] Stewart, R., Tuexen, M., Poon, K., Lei, P., and V. Yasevich, "Sockets API Extensions for the Stream Control Transmission Protocol (SCTP)", RFC 6458, DOI 10.17487/RFC6458, December 2011, <http://www.rfc-editor.org/info/rfc6458>.

[RHB15] Rajiullah, M., Hurtig, P., Brunstrom, A., Petlund, A., and M. Welzl, "An Evaluation of Tail Loss Recovery Mechanisms for TCP", ACM SIGCOMM CCR 45 (1), January 2015.

[RJ10] Ramachandran, S., "Web metrics: Size and number of resources", Google http://code.google.com/speed/articles/web-metrics.html, May 2010.

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

Authors' Addresses

Per Hurtig
Karlstad University
Universitetsgatan 2
Karlstad 651 88
Sweden

Phone: +46 54 700 23 35
Email: per.hurtig@kau.se

Anna Brunstrom
Karlstad University
Universitetsgatan 2
Karlstad 651 88
Sweden

Phone: +46 54 700 17 95
Email: anna.brunstrom@kau.se

Andreas Petlund
Simula Research Laboratory AS
P.O. Box 134
Lysaker 1325
Norway

Phone: +47 67 82 82 00
Email: apetlund@simula.no

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

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

TCPM WG
Internet Draft
Updates: 793
Intended status: Standards Track
Expires: April 2015

J. Touch
USC/ISI
Wes Eddy
MTI Systems

October 13, 2014

TCP Extended Data Offset Option
draft-ietf-tcpm-tcp-edo-01.txt

Status of this Memo

Copyright Notice

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 that have evolved. This document updates RFC 793 with an optional TCP extension to that space to support the use of multiple large options such as SACK with either TCP Multipath or TCP AO. It also explains why the initial SYN of a connection cannot be extending a single segment.

Table of Contents

1. Introduction

TCP's Data Offset 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 [Ch14].

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 middlebox misbehavior.

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. 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.

4. 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. The EDO option is organized as indicated in Figure 1 and Figure 2. When desired, initial SYN segments (i.e., those whose ACK bit is not set) use the EDO request option, which consists of the required Kind and Length fields only. Depending on capability and whether EDO is successfully negotiated, any other segments can use the EDO length option, which adds a Header_Length field (in network-standard byte order), indicating the length of the entire TCP header in 32-bit words. The codepoint value of the EDO Kind is EDO-OPT.

Figure 1 TCP EDO request option

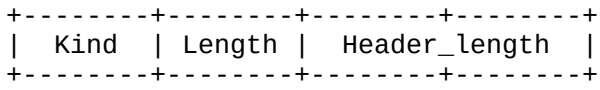


Figure 2 TCP EDO length option

EDO support is determined in both directions using a single exchange. An endpoint seeking to enable EDO support includes the EDO request option in the initial SYN. If receiver of that SYN agrees to support EDO, it responds with a null EDO length option in the SYN/ACK. A null EDO length option contains the same value as the DO field, i.e., it does not extend the TCP option space.

>> Connections using EDO MUST negotiate its availability during the initial three-way handshake.

>> An endpoint confirming EDO support MUST respond with a null EDO length option in its SYN/ACK.

The SYN/ACK uses the null EDO length option because it may not yet be safe to extend the option space in the reverse direction due to middlebox misbehavior (see Section 6.2). Extension of the SYN and SYN/ACK space is addressed as a separate option (see Section 7.7).

>> The EDO length option MAY be used only if confirmed when the connection transitions to the ESTABLISHED state, e.g., a client is enabled after receiving the null EDO length option in the SYN/ACK and the server is enabled after seeing a null or non-null EDO length option in the final ACK of the three-way handshake. If either of those segments lacks the EDO length option, the connection MUST NOT use EDO on any other segments.

>> Once enabled on a connection, all segments in both directions MUST include the EDO length option. Segments not needing extension MUST set the EDO length equal to the DO length.

Internet paths may vary after connection establishment, introducing misbehaving middleboxes (see Section 6.2). Using EDO on all segments in both directions allows this condition to be detected.

>> The EDO request option MAY occur in an initial SYN as desired (e.g., as expressed by the user/application), but MUST NOT be inserted in other segments. If the EDO request option is received in other segments, it MUST be silently ignored.

>> If EDO has not been negotiated and agreed, the EDO length option MUST be silently ignored on subsequent segments. The EDO length option MUST NOT be sent in an initial SYN segment, and MUST be silently ignored and not acknowledged if so received.

>> If EDO has been negotiated, any subsequent segments arriving without the EDO length 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 Data Offset for that segment.

>> The EDO length option MUST occur within the space indicated by the TCP Data Offset.

>> The EDO length 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 length option MUST be at least as large as the TCP Data Offset field of the segment in which they both appear. When the EDO length equals the DO length, the EDO option is present but it does not extend the option space. When the EDO length is invalid, the TCP segment MUST be silently dropped.

>> The EDO request option SHOULD be aligned on a 16-bit boundary and the EDO length 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, where EDO would be the first option processed, at which point the EDO length 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 length 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 length 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 length option to operate correctly.

5. TCP EDO Interaction with TCP

The following subsections describe how EDO interacts with the TCP specification [RFC793].

5.1. 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 6), 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.

5.2. TCP States and Transitions

TCP EDO does not alter the existing TCP state or state transition mechanisms.

5.3. TCP Segment Processing

TCP EDO alters segment processing during the TCP option processing step. Once detected, the TCP EDO length option overrides the TCP Data Offset field for all subsequent option processing. Option processing continues at the next option (if present) after the EDO length option.

5.4. Impact on TCP Header Size

The TCP EDO request option increases SYN header length by a minimum of 2 bytes. Currently popular SYN options total 19 bytes, which leaves more than enough room for the EDO request:

- SACK permitted (2 bytes in SYN, optionally 2 + 8N bytes after) [RFC2018][RFC6675]
- Timestamp (10 bytes) [RFC7323]
- Window scale (3 bytes) [RFC7323]
- MSS option (4 bytes) [RFC793]

Adding the EDO option would result in a total of 21 bytes of SYN option space. Subsequent segments would use 19 bytes of option space without any SACK blocks 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 [Ch14].

TCP EDO can also be negotiated in SYNs with either of the following large options:

- TCP-AO (authentication) (16 bytes) [RFC5925]
- Multipath TCP (12 bytes in SYN and SYN/ACK, 20 after) [RFC6824]

Including TCP-AO increases the SYN option space use to 37 bytes; with Multipath TCP the use is 33 bytes. When Multipath TCP is enabled with the typical options, later segments might require 39 bytes without SACK, thus effectively disabling the SACK option unless EDO is also supported on at least non-SYN segments.

The full combination of the above options (49 bytes including EDO) 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 [Ni14]. Even using "TS-permitted", the total space is still too large to support in the initial SYN without SYN option space extension [Br14][To14].

The EDO 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 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.

5.5. 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 option is mandatory on all TCP segments once negotiated, except the SYN and SYN/ACK of the three-way handshake to establish its support and the RST. A RST may lack the context to know that EDO is active on a connection.

>> The EDO length 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 option, the transmitted RST MUST NOT include the EDO length option.

5.6. 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.

6. 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

6.1. 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.

6.2. 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 modify TCP segment boundaries, might mix option information with user data if they did not support EDO. Such devices might also interfere with other TCP options such as TCP-AO. There are three types of such boxes:

- Those that process received options and transmit sent options separately, i.e., although they rewrite segments, they behave as TCP endpoints in both directions.

- Those that split segments, taking a received segment and emitting two or more segments with revised headers.

- Those that join segments, receiving multiple segments and emitting a single segment whose data is the concatenation of the components.

In all three cases, EDO is either treated as independent on different sides of such boxes or not. If independent, EDO would either be correctly terminated in either or both directions or disabled due to lack of SYN/ACK confirmation in either or both directions. Problems would occur only when TCP segments with EDO are combined or split while ignoring the EDO option. In the split case, the key concern is if the split happens within the option extension space or if EDO is silently copied to both segments without copying the corresponding extended option space contents. However, the most comprehensive study of these cases indicates that "although middleboxes do split and coalesce segments, none did so while passing unknown options" [Ho11].

Middleboxes that silently remove options they do not implement have been observed [Ho11]. Such boxes interfere with the use of the EDO length option in the SYN and SYN/ACK segments because extended option space would be misinterpreted as user data if the EDO option were removed, and this cannot be avoided. This is one reason that SYN and SYN/ACK extension requires alternate mechanisms (see Section 7.7). 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 length option be avoided on the SYN/ACK and that this option needs to be used on all segments once successfully negotiated.

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, such as IPsec, TCP-AO, etc. It is useful to note that such protection helps find only non-compliant components.

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.

7. 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.

7.1. 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.

7.2. 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.

7.3. 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.

7.4. 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.

7.5. LO/SLO

The TCP Long Option (LO, [Ed08]) is very similar to EDO, except that presence of LO results in ignoring the existing 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.

7.6. 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).

7.7. 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 [Ch14]. 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 [Br14][To14].

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.

8. 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.

9. Security Considerations

It is meaningless to have the Data Offset further exceed the position of the EDO data offset option.

>> When the EDO length option is present, the EDO length 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.

10. 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.

11. References

11.1. Normative References

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

[RFC793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

11.2. Informative References

[Al06] Allman, M., "TCPx2: Don’t Fence Me In", draft-allman-tcpx2-hack-00 (work in progress), May 2006.

[Br14] Briscoe, B., "Extended TCP Option Space in the Payload of an Alternative SYN", draft-briscoe-tcpm-syn-op-sis-02 (work in progress), September 2014.

[Ch14] Cheng, Y., Chu, J., and A. Jain, "TCP Fast Open", draft-ietf-tcpm-fastopen-10, September 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.

[Ni14] Nishida, Y., "A-PAWS: Alternative Approach for PAWS", draft-nishida-tcpm-apaws-01 (work in progress), June 2014.

[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.

[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.

[To14] Touch, J., T. Faber, "TCP SYN Extended Option Space Using an Out-of-Band Segment", draft-touch-tcpm-tcp-syn-ext-opt-01 (work in progress), September 2014.

[Yo11] Yourtchenko, A., "Introducing TCP Long Options by Invalid Checksum", draft-yourtchenko-tcp-loic-00 (work in progress), April 2011.

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

Authors' Addresses

Joe Touch
USC/ISI
4676 Admiralty Way
Marina del Rey, CA 90292-6695 USA

Phone: +1 (310) 448-9151
Email: touch@isi.edu

Wesley M. Eddy
MTI Systems
US

Email: wes@mti-systems.com

TCP Maintenance and Minor Extensions (TCPM) WG
Internet-Draft
Intended status: Experimental
Expires: October 18, 2015

I. Rhee
NCSU
L. Xu
UNL
S. Ha
NCSU
A. Zimmermann
L. Eggert
R. Scheffenegger
NetApp
April 16, 2015

CUBIC for Fast Long-Distance Networks
draft-zimmermann-tcpm-cubic-01

Abstract

CUBIC is an extension to the current TCP standards. The protocol differs from the current TCP standards only in the congestion window adjustment function in the sender side. In particular, it uses a cubic function instead of a linear window increase of the current TCP standards to improve scalability and stability under fast and long distance networks. BIC-TCP, a predecessor of CUBIC, has been a default TCP adopted by Linux since year 2005 and has already been deployed globally and in use for several years by the Internet community at large. CUBIC is using a similar window growth function as BIC-TCP and is designed to be less aggressive and fairer to TCP in bandwidth usage than BIC-TCP while maintaining the strengths of BIC-TCP such as stability, window scalability and RTT fairness. Through extensive testing in various Internet scenarios, we believe that CUBIC is safe for deployment and testing in the global Internet. The intent of this document is to provide the protocol specification of CUBIC for a third party implementation and solicit the community feedback through experimentation on the performance of CUBIC. We expect this document to be eventually published as an experimental RFC.

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/.

This Internet-Draft will expire on October 18, 2015.

Table of Contents

1. Introduction

The low utilization problem of TCP in fast long-distance networks is well documented in [K03][RFC3649]. This problem arises from a slow increase of congestion window following a congestion event in a network with a large bandwidth delay product (BDP). Our experience [HKLRX06] indicates that this problem is frequently observed even in the range of congestion window sizes over several hundreds of packets (each packet is sized around 1000 bytes) especially under a network path with over 100ms round-trip times (RTTs). This problem is equally applicable to all Reno style TCP standards and their variants, including TCP-RENO [RFC5681], TCP-NewReno [RFC6582], TCP-SACK [RFC2018], SCTP [RFC4960], TFRC [RFC5348] that use the same linear increase function for window growth, which we refer to collectively as Standard TCP below.

CUBIC [HRX08] is a modification to the congestion control mechanism of Standard TCP, in particular, to the window increase function of Standard TCP senders, to remedy this problem. It uses a cubic increase function in terms of the elapsed time from the last congestion event. While most alternative algorithms to Standard TCP uses a convex increase function where after a loss event, the window increment is always increasing, CUBIC uses both the concave and convex profiles of a cubic function for window increase. After a window reduction following a loss event, it registers the window size where it got the loss event as W_max and performs a multiplicative decrease of congestion window and the regular fast recovery and retransmit of Standard TCP. After it enters into congestion avoidance from fast recovery, it starts to increase the window using the concave profile of the cubic function. The cubic function is set to have its plateau at W_max so the concave growth continues until the window size becomes W_max. After that, the cubic function turns into a convex profile and the convex window growth begins. This style of window adjustment (concave and then convex) improves protocol and network stability while maintaining high network utilization [CEHRX07]. This is because the window size remains almost constant, forming a plateau around W_max where network utilization is deemed highest and under steady state, most window size samples of CUBIC are close to W_max, thus promoting high network utilization and protocol stability. Note that protocols with convex increase functions have the maximum increments around W_max and introduces a large number of packet bursts around the saturation point of the network, likely causing frequent global loss synchronizations.

Another notable feature of CUBIC is that its window increase rate is mostly independent of RTT, and follows a (cubic) function of the elapsed time since the last loss event. This feature promotes per-flow fairness to Standard TCP as well as RTT-fairness. Note that Standard TCP performs well under short RTT and small bandwidth (or small BDP) networks. Only in a large long RTT and large bandwidth (or large BDP) networks, it has the scalability problem. An alternative protocol to Standard TCP designed to be friendly to Standard TCP at a per-flow basis must operate must increase its window much less aggressively in small BDP networks than in large BDP networks. In CUBIC, its window growth rate is slowest around the inflection point of the cubic function and this function does not depend on RTT. In a smaller BDP network where Standard TCP flows are working well, the absolute amount of the window decrease at a loss event is always smaller because of the multiplicative decrease. Therefore, in CUBIC, the starting window size after a loss event from which the window starts to increase, is smaller in a smaller BDP network, thus falling nearer to the plateau of the cubic function where the growth rate is slowest. By setting appropriate values of the cubic function parameters, CUBIC sets its growth rate always no faster than Standard TCP around its inflection point. When the cubic function grows slower than the window of Standard TCP, CUBIC simply follows the window size of Standard TCP to ensure fairness to Standard TCP in a small BDP network. We call this region where CUBIC behaves like Standard TCP, the TCP-friendly region.

CUBIC maintains the same window growth rate independent of RTTs outside of the TCP-friendly region, and flows with different RTTs have the similar window sizes under steady state when they operate outside the TCP-friendly region. This ensures CUBIC flows with different RTTs to have their bandwidth shares linearly proportional to the inverse of their RTT ratio (the longer RTT, the smaller the share). This behavior is the same as that of Standard TCP under high statistical multiplexing environments where packet losses are independent of individual flow rates. However, under low statistical multiplexing environments, the bandwidth share ratio of Standard TCP flows with different RTTs is squarely proportional to the inverse of their RTT ratio [XHR04]. CUBIC always ensures the linear ratio independent of the levels of statistical multiplexing. This is an improvement over Standard TCP. While there is no consensus on a particular bandwidth share ratios of different RTT flows, we believe that under wired Internet, use of the linear share notion seems more reasonable than equal share or a higher order shares. HTCP [LS08] currently uses the equal share.

CUBIC sets the multiplicative window decrease factor to 0.2 while Standard TCP uses 0.5. While this improves the scalability of the protocol, a side effect of this decision is slower convergence

especially under low statistical multiplexing environments. This design choice is following the observation that the author of HSTCP [RFC3649] has made along with other researchers (e.g., [GV02]): the current Internet becomes more asynchronous with less frequent loss synchronizations with high statistical multiplexing. Under this environment, even strict MIMD can converge. CUBIC flows with the same RTT always converge to the same share of bandwidth independent of statistical multiplexing, thus achieving intra-protocol fairness. We also find that under the environments with sufficient statistical multiplexing, the convergence speed of CUBIC flows is reasonable.

In the ensuing sections, we provide the exact specification of CUBIC and discuss the safety features of CUBIC following the guidelines specified in [RFC5033].

2. Conventions

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. CUBIC Congestion Control

3.1. Window growth function

CUBIC maintains the acknowledgment (ACK) clocking of Standard TCP by increasing congestion window only at the reception of ACK. The protocol does not make any change to the fast recovery and retransmit of TCP-NewReno [RFC6582] and TCP-SACK [RFC2018]. During congestion avoidance after fast recovery, CUBIC changes the window update algorithm of Standard TCP. Suppose that W_max is the window size before the window is reduced in the last fast retransmit and recovery.

The window growth function of CUBIC uses the following function:

$$W(t) = C*(t-K)^3 + W_{max} \quad \text{(Eq. 1)}$$

where C is a constant fixed to determine the aggressiveness of window growth in high BDP networks, t is the elapsed time from the last window reduction,and K is the time period that the above function takes to increase W to W_max when there is no further loss event and is calculated by using the following equation:

$$K = \text{cubic\_root}(W_{max}*beta/C) \quad \text{(Eq. 2)}$$

where beta is the multiplication decrease factor. We discuss how we set C in the next Section in more details.

Upon receiving an ACK during congestion avoidance, CUBIC computes the window growth rate during the next RTT period using Eq. 1. It sets W(t+RTT) as the candidate target value of congestion window. Suppose that the current window size is cwnd. Depending on the value of cwnd, CUBIC runs in three different modes. First, if cwnd is less than the window size that Standard TCP would reach at time t after the last loss event, then CUBIC is in the TCP friendly region (we describe below how to determine this window size of Standard TCP in term of time t). Otherwise, if cwnd is less than W_max, then CUBIC is the concave region, and if cwnd is larger than W_max, CUBIC is in the convex region. Below, we describe the exact actions taken by CUBIC in each region.

3.2. TCP-friendly region

When receiving an ACK in congestion avoidance, we first check whether the protocol is in the TCP region or not. This is done as follows. We can analyze the window size of Standard TCP in terms of the elapsed time t. Using a simple analysis in [FHP00], we can analyze the average window size of additive increase and multiplicative decrease (AIMD) with an additive factor alpha and a multiplicative factor beta to be the following function:

$$(\alpha/2 * (2-\beta)/\beta * 1/p)^{0.5} \quad \text{(Eq. 3)}$$

By the same analysis, the average window size of Standard TCP with alpha 1 and beta 0.5 is $(3/2 * 1/p)^{0.5}$. Thus, for Eq. 3 to be the same as that of Standard TCP, alpha must be equal to $3*\beta/(2-\beta)$. As Standard TCP increases its window by alpha per RTT, we can get the window size of Standard TCP in terms of the elapsed time t as follows:

$$W_{tcp}(t) = W_{max}*(1-\beta) + 3*\beta/(2-\beta)* t/RTT \quad \text{(Eq. 4)}$$

If cwnd is less than W_tcp(t), then the protocol is in the TCP friendly region and cwnd SHOULD be set to W_tcp(t) at each reception of ACK.

3.3. Concave region

When receiving an ACK in congestion avoidance, if the protocol is not in the TCP-friendly region and cwnd is less than W_max, then the protocol is in the concave region. In this region, cwnd MUST be incremented by (W(t+RTT) - cwnd)/cwnd.

3.4. Convex region

When the window size of CUBIC is larger than W_max, it passes the plateau of the cubic function after which CUBIC follows the convex profile of the cubic function. Since cwnd is larger than the previous saturation point W_max, this indicates that the network conditions might have been perturbed since the last loss event, possibly implying more available bandwidth after some flow departures. Since the Internet is highly asynchronous, some amount of perturbation is always possible without causing a major change in available bandwidth. In this phase, CUBIC is being very careful by very slowly increasing its window size. The convex profile ensures that the window increases very slowly at the beginning and gradually increases its growth rate. We also call this phase as the maximum probing phase since CUBIC is searching for a new W_max. In this region, cwnd MUST be incremented by (W(t+RTT) - cwnd)/cwnd for each received ACK.

3.5. Multiplicative decrease

When a packet loss occurs, CUBIC reduces its window size by a factor of beta. Parameter beta SHOULD be set to 0.2.

```
W_max = cwnd;              // save window size before reduction
cwnd = cwnd * (1-beta);    // window reduction
```

A side effect of setting beta to a smaller value than 0.5 is slower convergence. We believe that while a more adaptive setting of beta could result in faster convergence, it will make the analysis of the protocol much harder. This adaptive adjustment of beta is an item for the next version of CUBIC.

3.6. Fast convergence

To improve the convergence speed of CUBIC, we add a heuristic in the protocol. When a new flow joins the network, existing flows in the network need to give up their bandwidth shares to allow the flow some room for growth if the existing flows have been using all the bandwidth of the network. To increase this release of bandwidth by existing flows, the following mechanism called fast convergence SHOULD be implemented.

With fast convergence, when a loss event occurs, before a window reduction of congestion window, a flow remembers the last value of W_max before it updates W_max for the current loss event. Let us call the last value of W_max to be W_last_max.

```
if (W_max < W_last_max){        // check downward trend
    W_last_max = W_max;         // remember the last W_max
    W_max = W_max*(2-beta)/2;   // further reduce W_max
} else {                        // check upward trend
    W_last_max = W_max          // remember the last W_max
}
```

This allows W_max to be slightly less than the original W_max. Since flows spend most of time around their W_max, flows with larger bandwidth shares tend to spend more time around the plateau allowing more time for flows with smaller shares to increase their windows.

4. Discussion

With a deterministic loss model where the number of packets between two successive lost events is always 1/p, CUBIC always operates with the concave window profile which greatly simplifies the performance analysis of CUBIC. The average window size of CUBIC can be obtained by the following function:

$$(C*(4-beta)/4/beta)^{0.25} * RTT^{0.75} / p^{0.75} \quad \text{(Eq. 5)}$$

With beta set to 0.2, the above formula is reduced to:

$$(C*3.8/0.8)^{0.25} * RTT^{0.75} / p^{0.75} \quad \text{(Eq. 6)}$$

We will determine the value of C in the following subsection using Eq. 6.

4.1. Fairness to standard TCP

In environments where standard TCP is able to make reasonable use of the available bandwidth, CUBIC does not significantly change this state.

Standard TCP performs well in the following two types of networks:

1. networks with a small bandwidth-delay product (BDP)

2. networks with a short RTT, but not necessarily a small BDP

CUBIC is designed to behave very similarly to standard TCP in the above two types of networks. The following two tables show the average window size of standard TCP, HSTCP, and CUBIC. The average window size of standard TCP and HSTCP is from [RFC3649]. The average window size of CUBIC is calculated by using Eq. 6 and CUBIC TCP friendly mode for three different values of C.

Loss Rate P	TCP	HSTCP	CUBIC (C=0.04)	CUBIC (C=0.4)	CUBIC (C=4)
10^-2	12	12	12	12	12
10^-3	38	38	38	38	66
10^-4	120	263	120	209	371
10^-5	379	1795	660	1174	2087
10^-6	1200	12279	3713	6602	11740
10^-7	3795	83981	20878	37126	66022
10^-8	12000	574356	117405	208780	371269

Response function of standard TCP, HSTCP, and CUBIC in networks with RTT = 100ms. The average window size W is in MSS-sized segments.

Table 1

Loss Rate P	Average TCP W	Average HSTCP W	CUBIC (C=0.04)	CUBIC (C=0.4)	CUBIC (C=4)
10^-2	12	12	12	12	12
10^-3	38	38	38	38	38
10^-4	120	263	120	120	120
10^-5	379	1795	379	379	379
10^-6	1200	12279	1200	1200	2087
10^-7	3795	83981	3795	6603	11740
10^-8	12000	574356	20878	37126	66022

Response function of standard TCP, HSTCP, and CUBIC in networks with RTT = 10ms. The average window size W is in MSS-sized segments.

Table 2

Both tables show that CUBIC with any of these three C values is more friendly to TCP than HSTCP, especially in networks with a short RTT where TCP performs reasonably well. For example, in a network with RTT = 10ms and p=10^-6, TCP has an average window of 1200 packets. If the packet size is 1500 bytes, then TCP can achieve an average rate of 1.44 Gbps. In this case, CUBIC with C=0.04 or C=0.4 achieves exactly the same rate as Standard TCP, whereas HSTCP is about ten times more aggressive than Standard TCP.

We can see that C determines the aggressiveness of CUBIC in competing with other protocols for the bandwidth. CUBIC is more friendly to the Standard TCP, if the value of C is lower. However, we do not

recommend to set C to a very low value like 0.04, since CUBIC with a low C cannot efficiently use the bandwidth in long RTT and high bandwidth networks. Based on these observations, we find C=0.4 gives a good balance between TCP-friendliness and aggressiveness of window growth. Therefore, C SHOULD be set to 0.4. With C set to 0.4, Eq. 6 is reduced to:

$$1.17 * RTT^{0.75} / p^{0.75} \quad \text{(Eq. 7)}$$

Eq. 7 is then used in the next subsection to show the scalability of CUBIC.

4.2. Using Spare Capacity

CUBIC uses a more aggressive window growth function than Standard TCP under long RTT and high bandwidth networks.

The following table shows that to achieve 10Gbps rate, standard TCP requires a packet loss rate of 2.0e-10, while CUBIC requires a packet loss rate of 3.4e-8.

Throughput(Mbps)	Average W	TCP P	HSTCP P	CUBIC P
1	8.3	2.0e-2	2.0e-2	2.0e-2
10	83.3	2.0e-4	3.9e-4	3.3e-4
100	833.3	2.0e-6	2.5e-5	1.6e-5
1000	8333.3	2.0e-8	1.5e-6	7.3e-7
10000	83333.3	2.0e-10	1.0e-7	3.4e-8

Required packet loss rate for Standard TCP, HSTCP, and CUBIC to achieve a certain throughput. We use 1500-byte packets and an RTT of 0.1 seconds.

Table 3

Our test results in [HKLRX06] indicate that CUBIC uses the spare bandwidth left unused by existing Standard TCP flows in the same bottleneck link without taking away much bandwidth from the existing flows.

4.3. Difficult Environments

CUBIC is designed to remedy the poor performance of TCP in fast long-distance networks. It is not designed for wireless networks.

4.4. Investigating a Range of Environments

CUBIC has been extensively studied by using both NS-2 simulation and test-bed experiments covering a wide range of network environments. More information can be found in [HKLRX06].

4.5. Protection against Congestion Collapse

In case that there is congestion collapse, CUBIC behaves likely standard TCP since CUBIC modifies only the window adjustment algorithm of TCP. Thus, it does not modify the ACK clocking and Timeout behaviors of Standard TCP.

4.6. Fairness within the Alternative Congestion Control Algorithm.

CUBIC ensures convergence of competing CUBIC flows with the same RTT in the same bottleneck links to an equal bandwidth share. When competing flows have different RTTs, their bandwidth shares are linearly proportional to the inverse of their RTT ratios. This is true independent of the level of statistical multiplexing in the link.

4.7. Performance with Misbehaving Nodes and Outside Attackers

This is not considered in the current CUBIC.

4.8. Responses to Sudden or Transient Events

In case that there is a sudden congestion, a routing change, or a mobility event, CUBIC behaves the same as Standard TCP.

4.9. Incremental Deployment

CUBIC requires only the change of TCP senders, and does not require any assistant of routers.

5. Security Considerations

This proposal makes no changes to the underlying security of TCP.

6. IANA Considerations

There are no IANA considerations regarding this document.

7. Acknowledgements

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

8. References

8.1. Normative References

[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.

[RFC3649] Floyd, S., "HighSpeed TCP for Large Congestion Windows", RFC 3649, December 2003.

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

[RFC5033] Floyd, S. and M. Allman, "Specifying New Congestion Control Algorithms", BCP 133, RFC 5033, August 2007.

[RFC5348] Floyd, S., Handley, M., Padhye, J., and J. Widmer, "TCP Friendly Rate Control (TFRC): Protocol Specification", RFC 5348, September 2008.

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

[RFC6582] Henderson, T., Floyd, S., Gurtov, A., and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 6582, April 2012.

8.2. Informative References

[CEHRX07] Cai, H., Eun, D., Ha, S., Rhee, I., and L. Xu, "Stochastic Ordering for Internet Congestion Control and its Applications", In Proceedings of IEEE INFOCOM , May 2007.

[FHP00] Floyd, S., Handley, M., and J. Padhye, "A Comparison of Equation-Based and AIMD Congestion Control", May 2000.

[GV02] Gorinsky, S. and H. Vin, "Extended Analysis of Binary Adjustment Algorithms", Technical Report TR2002-29, Department of Computer Sciences , The University of Texas at Austin , August 2002.

[HKLRX06] Ha, S., Kim, Y., Le, L., Rhee, I., and L. Xu, "A Step toward Realistic Performance Evaluation of High-Speed TCP Variants", International Workshop on Protocols for Fast Long-Distance Networks , February 2006.

[HRX08] Ha, S., Rhee, I., and L. Xu, "CUBIC: A New TCP-Friendly High-Speed TCP Variant", ACM SIGOPS Operating System Review , 2008.

[K03] Kelly, T., "Scalable TCP: Improving Performance in HighSpeed Wide Area Networks", ACM SIGCOMM Computer Communication Review , April 2003.

[LS08] Leith, D. and R. Shorten, "H-TCP: TCP Congestion Control for High Bandwidth-Delay Product Paths", Internet-draft draft-leith-tcp-htcp-06 , April 2008.

[XHR04] Xu, L., Harfoush, K., and I. Rhee, "Binary Increase Congestion Control for Fast, Long Distance Networks", In Proceedings of IEEE INFOCOM , March 2004.

Authors' Addresses

Injong Rhee
North Carolina State University
Department of Computer Science
Raleigh, NC 27695-7534
US

Email: rhee@ncsu.edu

Lisong Xu
University of Nebraska-Lincoln
Department of Computer Science and Engineering
Lincoln, NE 68588-01150
US

Email: xu@unl.edu

Sangtae Ha
University of Colorado at Boulder
Department of Computer Science
Boulder, CO 80309-0430
US

Email: sangtae.ha@colorado.edu

Alexander Zimmermann
NetApp
Sonnenallee 1
Kirchheim 85551
Germany

Phone: +49 89 900594712
Email: alexander.zimmermann@netapp.com

Lars Eggert
NetApp
Sonnenallee 1
Kirchheim 85551
Germany

Phone: +49 151 12055791
Email: lars@netapp.com

Richard Scheffenegger
NetApp
Am Euro Platz 2
Vienna 1120
Austria

Phone: +43 1 3676811 3146
Email: rs@netapp.com

TCP Maintenance and Minor Extensions (TCPM) WG
Internet-Draft
Intended status: Standards Track
Expires: May 14, 2015

A. Zimmermann
NetApp, Inc.
L. Schulte
Aalto University
C. Wolff
A. Hannemann
credativ GmbH

November 10, 2014

Detection and Quantification of Packet Reordering with TCP
draft-zimmermann-tcpm-reordering-detection-02

Abstract

This document specifies an algorithm for the detection and quantification of packet reordering for TCP. In the absence of explicit congestion notification from the network, TCP uses only packet loss as an indication of congestion. One of the signals TCP uses to determine loss is the arrival of three duplicate acknowledgments. However, this heuristic is not always correct, notably in the case when paths reorder packets. This results in degraded performance.

The algorithm for the detection and quantification of reordering in this document uses information gathered from the TCP Timestamps Option, the TCP SACK Option and its DSACK extension. When a reordering event is detected, the algorithm calculates a reordering extent by determining the number of segments the reordered segment was late with respect to its position in the sequence number space. Additionally, the algorithm computes a second reordering extent that is relative to the amount of outstanding data and thus provides a better estimation of the reordering delay when other sender state changes.

time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 14, 2015.

Table of Contents

1. Introduction

When the Transmission Control Protocol (TCP) [RFC0793] decides that the oldest outstanding segment is lost, it performs a retransmission and changes the sending rate [RFC5681]. This occurs either when the Retransmission Timeout (RTO) timer expires for a segment [RFC6298], or when three duplicate acknowledgments (ACKs) for a segment have been received (Fast Retransmit/Fast Recovery) [RFC5681]. The assumption behind Fast Retransmit is that non-congestion events that can cause duplicate ACKs to be generated (packet duplication, packet reordering and packet corruption) are infrequent. However, a number of Internet measurement studies have shown that packet reordering is not a rare phenomenon [Pax97], [BPS99], [BS02], [ZM04], [GPL04], [JIDKT07] and has negative performance implications on TCP [BA02], [ZKFP03].

From TCP's perspective, the result of packet reordering on the forward-path is the reception of out-of-order segments by the TCP receiver. In response to every received out-of-order segment, the TCP receiver immediately sends a duplicate ACK. (Note: [RFC5681] recommends that delayed ACKs not be used when the ACK is triggered by an out-of-order segment.) The sender side, if the number of consecutively received duplicate ACKs exceeds the duplicate acknowledgment threshold (DupThresh), retransmits the first unacknowledged segment [RFC5681] and continues with a loss recovery algorithm such as NewReno [RFC6582] or the Selective Acknowledgment (SACK) based loss recovery [RFC6675]. If a segment arrives due to reordering more than three segments (the default value of DupThresh [RFC5681]) too late at the TCP receiver, the sender is not able to distinguish this reordering event from a segment loss, resulting in an unnecessary retransmission and rate reduction.

Since DupThresh is defined in segments rather than bytes [RFC5681], TCP usually quantifies packet reordering in terms of segments. Informally, the reordering extent [RFC4737] is defined as the maximum distance in segments between the reception of a reordered segment and the earliest segment received with a larger sequence number. If a segment is received in-order, its reordering extent is undefined [RFC4737].

Another approach taken by this specification quantifies the reordering extend for a packet not only through an absolute value, but also through a measure that is relative to the amount of outstanding data, in an attempt to approximate a time-based measure. The presented scheme can thereby easily be adapted to the Stream Control Transmission Protocol (SCTP) [RFC2960], since SCTP uses congestion control algorithms similar to TCP.

Overall, this document describes mechanisms to detect reordering on the forward-path during a TCP connection, and provides these samples as an input for an additional reaction algorithm.

The remainder of this document is organized as follows. Section 3 provides a high-level description of the packet reordering detection mechanisms. In Section 4, the algorithm is specified. In Section 5, each step of the algorithm is discussed in detail. Section 6 provides a discussion of several design decisions of the algorithm. Section 7 discusses related work. Section 9 discusses security concerns.

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 [RFC2119].

The reader is expected to be familiar with the TCP state variables given in [RFC0793] (SEG.SEQ, SND.UNA), [RFC5681] (FlightSize), and [RFC6675] (DupThresh, SACK scoreboard). SND.FACK (forward acknowledgment) is used to describe the highest sequence number - plus one - that has been either cumulatively or selectively acknowledged by the receiver and subsequently seen by the sender [MM96]. Further, the term ’acceptable acknowledgment’ is used as defined in [RFC0793]. That is, an ACK that increases the connection’s cumulative ACK point by acknowledging previously unacknowledged data. The term ’duplicate acknowledgment’ is used as defined in [RFC6675], which is different from the definition of duplicate acknowledgment in [RFC5681].

This specification defines the four TCP sender states ’open’, ’disorder’, ’recovery’, and ’loss’ as follows. As long as no duplicate ACK is received and no segment is considered lost, the TCP sender is in the ’open’ state. Upon the reception of the first consecutive duplicate ACK, TCP will enter the ’disorder’ state. After receiving DupThresh duplicate ACKs, the TCP sender switches to the ’recovery’ state and executes standard loss recovery procedures like Fast Retransmit and Fast Recovery [RFC5681]. Upon a retransmission timeout, the TCP sender enters the ’loss’ state. The

'recovery' state can only be reached by a transition from the 'disorder' state, the 'loss' state can be reached from any other state.

3. Basic Concepts

The following specification depends on the TCP Timestamps option [RFC1323], the TCP Selective Acknowledgment (SACK) [RFC2018] option and the latter's Duplicate Selective Acknowledgment (DSACK) extension [RFC2883]. The reader is assumed to be familiar with the algorithms specified in these documents.

Reordering is quantified by an absolute and a relative reordering extent. If a hole in the SACK scoreboard of the TCP sender is closed either cumulatively by an acceptable ACK or selectively by a new SACK, then the absolute reordering extent is computed as the number of segments in the SACK scoreboard between the sequence number of the reordered segment and the highest selectively or cumulatively acknowledged sequence number. The relative reordering extent is then computed as the ratio between the absolute reordering extent and the FlightSize stored when entering the 'disorder' state.

If the hole that was closed in the SACK scoreboard corresponds to a segment that was not retransmitted, or if the retransmission of such a segment can be determined as a spurious retransmission by means of the Eifel detection algorithm [RFC3522], then the calculated reordering extent is immediately valid. Otherwise, if the verification of the Eifel detection algorithm has not been possible, the reordering extent is stored for a possibly subsequent DSACK. If no such DSACK is received in the next two round-trip times (RTTs), the reordering extents are discarded.

4. The Algorithm

Given that usually both the Nagle algorithm [RFC0896] [RFC1122] and the TCP Selective Acknowledgment Option [RFC2018] are enabled, a TCP sender MAY employ the following algorithm to detect and quantify the current perceived packet reordering in the network.

Without the Nagle algorithm, there is no straight forward way to accurately calculate the number of outstanding segments in the network (and, therefore, no good way to derive an appropriate reordering extent) without adding state to the TCP sender. A TCP connection that does not employ the Nagle algorithm SHOULD NOT use this methodology.

If a TCP sender implements the following algorithm, the implementation MUST follow the various specifications provided in

Sections 4.1 to 4.8. The algorithm MUST be executed *before* the Transmission Control Block or the SACK scoreboard have been updated by another loss recovery algorithm.

4.1. Initialization During Connection Establishment

After the completion of the TCP connection establishment, the following state variables MUST be initialized in the TCP transmission control block:

(C.1) The variable Dsack, which indicates whether a DSACK has been received so far, and the data structure Samples, which stores the computed reordering extents, MUST be initialized as:

```
Dsack = false
Samples = []
```

(C.2) If the TCP Timestamps option [RFC1122] has been negotiated, then the variable Timestamps MUST be activated and the data structure Retrans_TS, which stores the value of the TSval field of the retransmissions sent during Fast Recovery, MUST be initialized. Additionally, the data structure Retrans_Dsack MAY be used in order to detect reordering longer than RTT with Timestamps and DSACK:

```
Timestamps = true
Retrans_TS = []
Retrans_Dsack = []
```

Otherwise, the Timestamps-based detection SHOULD be deactivated:

```
Timestamps = false
```

4.2. Receiving Acknowledgments

For each received ACK that either a) carries SACK information, *or* b) is a full ACK that terminates the current Fast Recovery procedure, *or* c) is an acceptable ACK that is received immediately after a duplicate ACK, execute steps (A.1) to (A.4), otherwise skip to step (A.4).

(A.1) If a) the ACK carries new SACK information, *and* b) the SACK scoreboard is empty (i.e., the TCP sender has received no SACK information from the receiver), then the TCP sender MUST save the amount of current outstanding data:

```
FlightSizePrev = FlightSize
```

(A.2) If the received ACK either a) cumulatively acknowledges at most SMSS bytes, *or* b) selectively acknowledges at most SMSS bytes in the sequence number space in the SACK scoreboard, then:

The TCP sender MUST execute steps (S.1) to (S.4)

(A.3) If a) Timestamps == false *and* b) the received ACK carries a DSACK option [RFC2883] and the segment identified by the DSACK option can be marked according to step (A.1) to (A.4) of [RFC3708] as a valid duplicate, then:

The TCP sender MUST execute steps (D.1) to (D.3)

(A.4) The TCP sender MUST terminate the processing of the ACK by this algorithm and MUST continue with the default processing of the ACK.

4.3. Receiving Acknowledgment Closing Hole

(S.1) If (a) the newly cumulatively or selectively acknowledged segment SEG is a retransmission *and* b) both equations Dsack == false and Timestamps == false hold, then the TCP sender MUST skip to step (A.4).

(S.2) Compute the relative and absolute reordering extent ReorExtR, ReorExtA:

The TCP sender MUST execute steps (E.1) to (E.4)

(S.3) If a) the newly acknowledged segment SEG was not retransmitted before *or* b) both equations Timestamps == true and Retrans_TS[SEG.SEQ] > ACK.TSecr hold, i.e., the ACK acknowledges the original transmission and not a retransmission, then hand over the reordering extents to an additional reaction algorithm:

The TCP sender MUST execute step (P)

(S.4) If a) the previous step (S.3) was not executed *and* b) both equations Dsack == true and Timestamps == false hold, save the reordering extents for the newly acknowledged segment SEG for at least two RTTs:

```
Samples[SEG.SEQ].ReorExtR = ReorExtR
Samples[SEG.SEQ].ReorExtA = ReorExtA
```

(S.5) If a) the newly acknowledged segment SEG was retransmitted before exactly once *and* b) both equations Dsack == true and Timestamps == true hold *and* c) Retrans_TS[SEG.SEQ] == ACK.TSecr, then save FlightSizePrev for this segment in order to be calculate the metrics in case a DSACK arrives, i.e. reordering delay is greater than RTT:

```
Retrans_Dsack[SEG.SEQ] = FlightSizePrev
```

4.4. Receiving Duplicate Selective Acknowledgment

(D.1) If no DSACK has been received so far, the sender MUST set:

```
Dsack = true
```

(D.2) If a) the previous step (D.1) was not executed *and* a reordering extent was calculated for the segment SEG identified by the DSACK option, then the TCP sender MUST restore the values of the variables ReorExtR and ReorExtA and delete the corresponding entries in the data structure:

```
ReorExtR = Samples[SEG.SEQ].ReorExtR
ReorExtA = SAMPLES[SEG.SEQ].ReorExtA
```

(D.3) If a) step (D.1) was not executed *and* b) FlightSizePrev was saved in step (S.4) for the segment, then the TCP sender MUST calculate the reordering extent for the segment with the E series of steps by using the FlightSizePrev saved for this segment and afterwards delete the corresponding entries:

```
FlightSizePrev_saved = Retrans_Dsack[SEG.SEQ]
```

(D.4) Hand the new reordering extents over to an additional reaction algorithm:

```
The TCP sender SHOULD execute step (P)
```

4.5. Reordering Extent Computation

(E.1) SEG.SEQ is the sequence number of the newly cumulatively or selectively acknowledged segment SEG.

(E.2) SND.FACK is the highest either cumulatively or selectively acknowledged sequence number so far plus one.

(E.3) The TCP sender MUST compute the absolute reordering extent ReorExtA as

```
ReorExtA = (SND.FACK - SEG.SEQ) / SMSS
```

(E.4) The TCP sender MUST compute the relative reordering extent ReorExtR as

```
ReorExtR= ReorExtA * (SMSS / FlightSizePrev)
```

4.6. Retransmitting Segment

If the TCP Timestamps option [RFC1323] is used to detect packet reordering, the TCP sender must save the TCP Timestamps option of all retransmitted segments during Fast Recovery.

(RET) If a) a segment SEG is retransmitted during Fast Recovery, *and* b) the equation Timestamps = true holds, the TCP sender MUST save the value of the TSval field of the retransmitted segment:

```
Retrans_TS[SEG.SEQ] = SEG.TSval
```

4.7. Placeholder for Response Algorithm

(P) This is a placeholder for an additional reaction algorithm that takes further action using the results of this algorithm, for example, the adjustment of the DupThresh based on relative and absolute reordering extent ReorExtR and ReorExtA.

4.8. Retransmission Timeout

The expiration of the retransmission timer should be interpreted as an indication of a change in path characteristics, and the TCP sender should consider all saved reordering extents as outdated and delete them.

(RTO) If an retransmission timeout (RTO) occurs, a TCP sender SHOULD reset the following variables:

```
Samples = []
Retrans_TS = []
FlightSizePrev = 0
```

5. Protocol Steps in Detail

The reception of an ACK represents the starting point for the detection scheme above. For each received SACK, DSACK or acceptable ACK that prompts the TCP sender to enter the ’disorder’ state, to remain in the ’disorder’ state or to leave either the ’disorder’ or ’recovery’ states towards the ’open’ state, steps (A.1) to (A.4) are

performed. All other received ACKs are not relevant for the detection of packet reordering and can be ignored. If the TCP sender changes from the 'open' to the 'disorder' state due to the reception of a duplicate ACK (i.e., the SACK scoreboard is empty and an ACK arrives carrying new SACK information), the current amount of outstanding data, FlightSize, is stored for the subsequent calculation of the relative reordering extent (step (A.1)).

Whenever a received acceptable ACK or SACK closes a hole in the sequence number space of the SACK scoreboard either partially or completely, this is an indication of packet reordering in the network (step (A.2)). The prerequisite for an accurate quantification of the reordering is that only one segment is newly acknowledged (maximum SMSS bytes of data). If more than one segment per ACK is acknowledged, either by reordering on the reverse path or the loss of ACKs, the order in which the segments have been received by the TCP receiver is no longer accurately determinable so that in this case a reordering extent is not calculated. Finally, if the received ACK carries a DSACK option that identifies a segment that was retransmitted only once, then this is sufficient to conclude reordering (step (A.3)), so that a previously calculated reordering extent can be passed to another algorithm (steps (D.3) and (P)).

With just the information provided by the ACK field or SACK information above SND.UNA, the TCP sender is unable to distinguish whether the ACK that finally acknowledges retransmitted data (either cumulatively or selectively) was sent in response to the original segment or a retransmission of the segment. This is described as the retransmission ambiguity problem in [KP87]. Therefore, the detection and quantification of reordering depends on other means to distinguish between acknowledgments for transmission and retransmission to detect if a retransmission was spurious. If neither a DSACK has been received (Dsack == false) so far nor the TCP Timestamps option has been enabled on connection establishment (Timestamps == false) then there is no possibility for the TCP sender to identify spurious retransmissions. Hence, the processing of the received ACK by the detection algorithm must be terminated for retransmitted segments (step (S.1)). Otherwise, if the segment that corresponds to the closed hole in the sequence number space of the SACK scoreboard has not been retransmitted or the retransmission can be identified by the Eifel detection algorithm [RFC3522] as a spurious retransmission, the previously calculated reordering extent is valid (step (S.2)) and an additional reaction algorithm can be executed (steps (S.3) and (P)).

For the use of the Eifel detection it is necessary to store the TCP Timestamps option of all retransmissions sent during Fast Recovery (step (Ret)). However, if the use of the Eifel detection algorithm

is not possible (Timestamps == false), the extent of a possible reordering is stored for the possibility of a subsequent arrival of a DSACK (step (P.4)). If no such DSACK is received in the next two round-trip times, the reordering extent is discarded. Since the DSACK extension is not negotiated during connection establishment [RFC2883], the reordering extent is only stored if a DSACK was previously received for the TCP connection (DSACK == true, step (D.1)).

Regardless of whether packet reordering is detected by using the SACK-based methodology, the DSACK-based methodology, or the TCP Timestamps option, quantification of the reordering will always be done when closing a hole in the sequence number space of the SACK scoreboard (step (A.2), step (P.2)). The only difference is the time of detection, which is in the case of DSACK-based methodology at least one RTT after the time of the quantification. The absolute reordering extent ReorExtA results from the number of segments in the SACK scoreboard between the sequence number of the newly acknowledged segment and the highest either cumulatively or selectively acknowledged sequence number so far plus one (SND.FACK) (step (E.3)).

In the case that the reordering delay is longer than RTT, the reordering can not be detected by timestamps or DSACK alone, but both algorithms are needed: when a packet is retransmitted, but no reordering could be detected when it was acknowledged, then it might be possible that a DSACK arrives for this packet. Then, the reordering extent was longer than RTT and the reordering extent has to be calculated at the point in time the DSACK arrives (step D.3). Therefore, we save the FlightSizePrev for a retransmitted segment when it is acked and no reordering is detected (step S.5).

It is worth noting that the absolute reordering extent includes all segments (bytes) between the closed hole and the highest acknowledged sequence number so far, i.e., it also includes segments (bytes) that are not selectively acknowledged. The reason is that if packet reordering is considered from a temporal perspective, it is irrelevant whether there are lost segments or not. The important fact is that the lost segments have been sent after the delayed segment and before the highest acknowledged segment, which is expressed by the metric. In step (E.4), the relative reordering extent ReorExtR is then calculated by the ratio between the absolute reordering extent ReorExtA and the amount of outstanding data stored by step (A.1).

6. Discussion of the Algorithm

The focus of the following discussion is on the quantification of reordering by the relative reordering extent and to elaborate on possible sources of error, which may lead to an inaccurate detection and quantification of reordering in the network.

6.1. Reasoning for the Relative Reordering Extent

A problem that arises with the way of quantifying reordering solemnly by the absolute reordering extent is that even in the presence of constant reordering, reordering extents may vary if the transmission rate of the TCP sender changes. Therefore, by using a DupThresh that directly reflects the measured reordering extent, spurious retransmissions cannot be fully avoided.

The following example illustrates this issue. Assume a path with a reordering probability of 1%, a reordering delay of 20 ms, and a bottleneck bandwidth of 3 Mb/s. Because segments that are delayed by reordering arrive 20 ms too late, the TCP receiver can receive a maximum of ((20 * 3 * 10^3) / 8) = 7500 bytes out-of-order before the reordered segment arrives. Hence, with a Sender Maximum Segment Size (SMSS) of 1460 bytes, the largest possible reordering extent is close to 5 segments. If the bottleneck bandwidth changes from 3 Mb/s to 4 Mb/s, the maximum reordering extent will increase to 7 segments, although the reordering delay remains constant.

This simple example shows that even with constant reordering, spurious retransmissions cannot be completely avoided if the DupThresh directly reflects the reordering extent. On the other hand, the reordering extent and the resulting DupThresh can sometimes also be much too high and do not correspond to the actual packet reordering present on the path. For example, a slow start overshoot [Hoe96], [MM96], [MSMO97] at the end of slow start might induce such a problem.

An obvious solution to the problem would be to quantify packet reordering not by calculating a reordering extent, but by using the reordering late time offset [RFC4737]. Since the reordering late time offset is not specified in segments but captures the difference between the expected and actual reception time of a reordered segment, this way of quantifying reordering is independent of the current transmission rate. Disadvantages of this approach are however a higher complexity and a worse integration into the TCP specification, since an implementation would require additional timers, whereas TCP itself is self-clocked.

6.2. Calculation of the Relative Reordering Extent

Generally, the characteristics of a relative reordering extent should be that if packet reordering on a path is constant in terms of rate and delay, the relative reordering extent should also be constant, regardless of the current transmission rate of the TCP sender. The scheme proposed in this document is to calculate the relative reordering by getting the ratio between absolute reordering (the amount of data the reordered segment was received too late) and the amount of outstanding data stored when TCP sender was entering the 'disorder' state (the maximum amount of data a reordered segment can be received too late). Therefore, the relative reordering extent expresses the portion of currently outstanding data that is selectively acknowledged before the reordered segment is cumulatively acknowledged. If the transmission rate changes, the absolute reordering extent changes as well, but together with the amount of outstanding data, and hence the relative reordering extent stays constant.

A characteristic of the calculation of the relative reordering extent on the basis of currently outstanding amount of data is that the FlightSize reflects the bandwidth-delay-product and not the transmission rate. As a consequence, the relative reordering extent is not independent of the RTT. If the RTT of the communication path changes, the amount of outstanding data changes as well, but the absolute reordering extent remains constant. Hence, the relative reordering extent adapts. In principle it is possible to design an algorithm to compute the relative reordering extent independently of the RTT and to reflect only the characteristics of packet reordering of the path. But since the calculation would be far from trivial and introducing more complexity, this is considered to be future research.

6.3. Persistent Reception of Selective Acknowledgments

Especially on paths with a high bandwidth-delay-product, it is possible that even with a minor packet reordering, several segments in a single window of data are delayed. If, in addition, the sequence numbers of those segments are widely spaced in the sequence number space and the delay caused by packet reordering is sufficiently high, this might lead to a constant reception of out-of-order data. Hence, for each received segment, regardless of whether a hole in the sequence number space of the receive window is closed or not, an ACK is sent that carries SACK information. From TCP sender's perspective, this persistent receiving of new SACK information leads to the situation that the TCP sender enters the 'disorder' state when receiving the first SACK and never leaves it

again during the connection lifetime if no segment is lost in between.

In case of the above reordering detection and quantification scheme, the persistent reception of SACK blocks causes the amount of outstanding data, which is stored when the TCP sender enters the 'disorder' state, to never be updated, since FlightSize is only saved in step (A.1) when the SACK scoreboard is empty. If the transmission rate of the TCP sender, and therefore also the maximum amount of data a reordered segment can be received too late, changes significantly during its stay in the 'disorder' state, the actual amount of reordering is not accurately determined by the relative reordering extent. A decrease of the transmission rate would result in an overestimation of the reordering extent and vice versa.

A simple solution to the problem would be to store the maximum offset in terms of sequence number space by which a reordered segment can be received too late only when entering the 'disorder' state, but individually for every potentially reordered segment, that is, for every hole in the sequence number space of the SACK scoreboard. (Note: The maximum offset in terms of sequence number space by which a reordered segment can be received too late is strictly speaking the amount of data that have been transmitted later than the reordered segment. This amount of data can only be expressed by FlightSize within the 'open' state and not within the 'disorder' state, since the cumulative ACK point may not advance).

The problem with this simple idea is that for a new hole in the SACK scoreboard, it is not possible to determine whether it is a result of packet reordering or loss, and therefore it results in increased memory usage (to store the amount of data for each hole). Additionally, the packet reordering would be inaccurately quantified if the transmission rate changes significantly for a short amount of time. For example, if the amount of outstanding data is low when entering the 'disorder' state is entered, the execution of Careful Extended Limited Transmit (as described in [I-D.zimmermann-tcpm-reordering-reaction] [RFC4653]) leads to a significant short-term change of the transmission rate. When the amount of data by which the reordering segment can be delayed is determined individually for every new hole, it leads to an overestimation of the relative reordering extent, since the maximum amount of data possible is 'artificially' reduced by Careful Extended Limited Transmit.

A solution to this problem is to store the maximum offset in terms of sequence number space by which a reordered segment can be received too late not for every segment individually (which does not guarantee an accurate calculation of the relative reordering extent) but only

sufficiently often, e.g., once per RTT. The identification of what frequency would be adequate, though, is neither trivial nor universally applicable, since a concrete solution depends on the transmission behavior of the used TCP in the 'disorder' state and whether it is more beneficial for an additional reordering response algorithm to over- or underestimate the packet reordering on the path. If, for example, TCP-aNCR [I-D.zimmermann-tcpm-reordering-reaction] is used as additional reordering response algorithm, the maximum offset in terms of sequence number space by which a reordered segment can be received too late is not only stored when entering the 'disorder' state but also updated every RTT (every cwnd worth of data transmitted without a loss) while the TCP sender stays in the 'disorder' state.

6.4. Unreliable ACK reception

ACK loss and ACK reordering are a cause for inaccuracies in samples.

6.5. Packet Duplication

Although the problem of packet duplication in today's Internet [JIDKT07], [MMMR08] is negligible, it may happen in rare cases that segments on the path to the TCP receiver are duplicated. If a segment is duplicated on the path, the first incoming segment causes the receiver to send either an acceptable ACK or a SACK, depending on whether the segment is the next expected one or not. Each subsequent identical segment then causes either a duplicate ACK or a DSACK, respectively, depending on whether the DSACK extension [RFC3708] is implemented or not.

If by a combination of packet loss and packet duplication the case occurs that a Fast Retransmit for a lost segment is duplicated on the path, the TCP sender is not able to distinguish this from packet reordering. The first received ACK closes a hole in the sequence number space of the SACK scoreboard, while the second received ACK is a valid DSACK. Although both cases are indistinguishable from a theoretical point of view, the TCP sender can take measures to ensure as far as possible that the DSACK received was not the result of packet duplication.

For this purpose, step (A.3) of the above detection method checks via the steps (A.1) to (A.4) of [RFC3708] whether the segment identified by the DSACK option is marked as a valid duplicate. Unfortunately, the steps of [RFC3708] do not check that more DSACKs have been received than retransmissions have been sent, which is a characteristic of suffering both packet reordering and packet duplication at the same time. By simply counting the received

DSACKs, for example, as additional step (A.5) in [RFC3708], this corner case can be covered as well.

7. Related Work

Because of retransmission ambiguity problem [KP87], which describes TCP sender's inability to to distinguish whether the first acceptable ACK that arrives after a retransmit was sent in response to the original transmit or the retransmit, two different approaches can generally be taken to detect and quantify packet reordering. First, for transmissions (non-retransmitted segments), the detection is usually conducted by detecting a closed hole in sequence number space of the SACK scoreboard. Second, for retransmissions, the detection of packet reordering is accompanied by the detection of the spurious Fast Retransmits.

Within the IETF, several proposals have been published in the RFC series to detect and quantify packet reordering. With [RFC4737] the IPPM Working Group [IPPM] defines several metrics to evaluate whether a network path has maintained packet order on a packet-by-packet basis. [RFC4737] gives concrete, well-defined metrics, along with a methodology for applying the metric to a generic packet stream. The metric discussed in this document has a different purpose from the IPPM metrics; this document discusses a TCP specific reordering metric calculated on the TCP sender's SACK scoreboard.

Besides the IPPM work, several other proposals have been developed to detect spurious retransmissions with TCP. The Eifel detection algorithm [RFC3522] uses the TCP Timestamps option to determine whether the ACK for a given retransmit is for the original transmission or a retransmission. The F-RTO scheme [RFC5682] slightly alters TCP's sending pattern immediately following a retransmission timeout to indicate whether the retransmitted segment was needed. Finally, the DSACK-based algorithm [RFC3708] uses the TCP SACK option [RFC2018] with the DSACK extension [RFC2883] to identify unnecessary retransmissions. The mechanism for detecting packet reordering outlined in this document rely on the detection schemes of those documents (except F-RTO that only works for spurious retransmits triggered by TCP's retransmission timer), although they do not provide metrics for the reordering extent whereas the algorithm described in this document does.

RR-TCP [ZKFP03] describes a reordering detection and quantification scheme that is also based on holes in the sequence number space of the SACK scoreboard and the reception of DSACKs. For every hole in the SACK scoreboard, RR-TCP calculates a reordering extent. If the segment was retransmitted before an ACK was received, it waits for a DSACK that proves that the segment was spuriously retransmitted. The

reordering sample in such a case is the mean between the sample calculated due to the hole in the sequence number space and the sample calculated in responding to the received DSACK.

The Linux kernel [Linux] implements a reordering detection based on SACK, DSACK and TCP Timestamps option as well. The detection and quantification of non-retransmitted segments with SACK or for retransmitted segments with TCP Timestamps option operates much like the scheme described in this document, with the exception of the DSACK detection. First, Linux does not store any information (e.g., reordering extent) below the cumulative ACK point, so that DSACKs below the cumulative ACK point are ignored (for the purpose for reordering quantification). Second, Linux also does not store any information about a possible reordering event when a hole in the sequence number space of the SACK scoreboard is closed. Therefore, for a DSACK reporting a duplicate above the cumulative ACK, Linux needs to approximate the reordering on arrival of a DSACK by the distance between the DSACK and the highest selectively acknowledged segment.

8. IANA Considerations

This memo includes no request to IANA.

9. Security Considerations

The described algorithm neither improves nor degrades the current security of TCP, since this document only detects and quantifies reordering and does not change the TCP behavior. General security considerations for SACK based loss recovery are outlined in [RFC6675].

10. Acknowledgments

The authors thank the flowgrind [Flowgrind] authors and contributors for their performance measurement tool, which give us a powerful tool to analyze TCP’s congestion control and loss recovery behavior in detail.

11. References

11.1. Normative References

[I-D.zimmermann-tcpm-reordering-reaction]
Zimmermann, A., Schulte, L., Wolff, C., and A. Hannemann, "An adaptive Robustness of TCP to Non-Congestion Events", draft-zimmermann-tcpm-reordering-reaction-01 (work in progress), November 2013.

[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.

[RFC1323] Jacobson, V., Braden, B., and D. Borman, "TCP Extensions for High Performance", RFC 1323, May 1992.

[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.

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

[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.

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

11.2. Informative References

[BA02] Blanton, E. and M. Allman, "On Making TCP More Robust to Packet Reordering", ACM Computer Communication Review vol.32, no. 1, pp. 20-30, January 2002.

[BPS99] Bennett, J., Partridge, C., and N. Shectman, "Packet reordering is not pathological network behavior", IEEE/ACM Transactions on Networking vol. 7, no. 6, pp. 789-798, December 1999.

[BS02] Bellardo, J. and S. Partridge, "Measuring Packet Reordering", Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment (IMW'02) pp. 97-105, November 2002.

[Flowgrind] "Flowgrind Home Page", <http://www.flowgrind.net>.

[GPL04] Gharai, L., Perkins, C., and T. Lehman, "Packet Reordering, High Speed Networks and Transport Protocol Performance", Proceedings of the 13th IEEE International Conference on Computer Communications and Networks (ICCCN'04) pp. 73-78, October 2004.

[Hoe96] Hoe, J., "Improving the Start-up Behavior of a Congestion Control Scheme for TCP", Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'96) pp. 270-280, August 1996.

[IPPM] "IP Performance Metrics (IPPM) Working Group", <http://www.ietf.org/html.charters/ippm-charter.html>.

[JIDKT07] Jaiswal, S., Iannaccone, G., Diot, C., Kurose, J., and D. Towsley, "Measurement and Classification of Out-of-Sequence Packets in a Tier-1 IP Backbone", IEEE/ACM Transactions on Networking vol. 15, no. 1, pp. 54-66, February 2007.

[KP87] Karn, P. and C. Partridge, "Improving Round-Trip Time Estimates in Reliable Transport Protocols", ACM SIGCOMM Computer Communication Review vol. 17, no. 5, pp. 2-7, November 1987.

[Linux] "The Linux Project", <http://www.kernel.org>.

[MMMR08] Mellia, M., Meo, M., Muscariello, L., and D. Rossi, "Passive analysis of TCP anomalies", Computer Networks vol. 52, no. 14, pp. 2663-2676, October 2008.

[MSMO97] Mathis, M., Semke, J., Mahdavi, J., and T. Ott, "The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm", ACM SIGCOMM Computer Communication Review vol. 27, no. 3, pp. 67-82, July 1997.

[Pax97] Paxson, V., "End-to-End Internet Packet Dynamics", IEEE/ACM Transactions on Networking vol. 7, no.3, pp. 277-292, June 1997.

[RFC0896] Nagle, J., "Congestion control in IP/TCP internetworks", RFC 896, January 1984.

[RFC1122] Braden, R., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, October 1989.

[RFC2960] Stewart, R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang, L., and V. Paxson, "Stream Control Transmission Protocol", RFC 2960, October 2000.

[RFC4653] Bhandarkar, S., Reddy, A., Allman, M., and E. Blanton, "Improving the Robustness of TCP to Non-Congestion Events", RFC 4653, August 2006.

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

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

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

[RFC6582] Henderson, T., Floyd, S., Gurtov, A., and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 6582, April 2012.

[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.

[ZKFP03] Zhang, M., Karp, B., Floyd, S., and L. Peterson, "RR-TCP: A Reordering-Robust TCP with DSACK", Proceedings of the 11th IEEE International Conference on Network Protocols (ICNP'03) pp. 95-106, November 2003.

[ZM04] Zhou, X. and P. Mieghem, "Reordering of IP Packets in Internet", Lecture Notes in Computer Science vol. 3015, pp. 237-246, April 2004.

Appendix A. Changes from previous versions of the draft

This appendix should be removed by the RFC Editor before publishing this document as an RFC.

A.1. Changes from draft-zimmermann-tcpm-reordering-detection-01

- Moved reasoning for relative reordering extent to discussion
- Extended algorithm for calculation of reordering extents greater than RTT (steps C.2, S.5 and D.3)
- Remove reverse-path reordering from intro

A.2. Changes from draft-zimmermann-tcpm-reordering-detection-00

- Improved the wording throughout the document.
- Replaced and updated some references.

Authors' Addresses

Alexander Zimmermann
NetApp, Inc.
Sonnenallee 1
Kirchheim 85551
Germany

Phone: +49 89 900594712
Email: alexander.zimmermann@netapp.com

Lennart Schulte
Aalto University
Otakaari 5 A
Espoo 02150
Finland

Phone: +358 50 4355233
Email: lennart.schulte@aalto.fi

Carsten Wolff
credativ GmbH
Hohenzollernstrasse 133
Moenchengladbach 41061
Germany

Phone: +49 2161 4643 182
Email: carsten.wolff@credativ.de

Arnd Hannemann
credativ GmbH
Hohenzollernstrasse 133
Moenchengladbach 41061
Germany

Phone: +49 2161 4643 134
Email: arnd.hannemann@credativ.de

TCP Maintenance and Minor Extensions (TCPM) WG
Internet-Draft
Obsoletes: 4653 (if approved)
Intended status: Experimental
Expires: May 14, 2015

A. Zimmermann
NetApp, Inc.
L. Schulte
Aalto University
C. Wolff
A. Hannemann
credativ GmbH

November 10, 2014

Making TCP Adaptively Robust to Non-Congestion Events
draft-zimmermann-tcpm-reordering-reaction-02

Abstract

This document specifies an adaptive Non-Congestion Robustness (aNCR) mechanism for TCP. In the absence of explicit congestion notification from the network, TCP uses only packet loss as an indication of congestion. One of the signals TCP uses to determine loss is the arrival of three duplicate acknowledgments. However, this heuristic is not always correct, notably in the case when paths reorder packets. This results in degraded performance.

TCP-aNCR is designed to mitigate this performance degradation by adaptively increasing the number of duplicate acknowledgments required to trigger loss recovery, based on the current state of the connection, in an effort to better disambiguate true segment loss from segment reordering. This document specifies the changes to TCP and TCP-NCR (on which this specification is build on) and discusses the costs and benefits of these modifications.

Status of This Memo

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

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

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

This Internet-Draft will expire on May 14, 2015.

Table of Contents

1. Introduction

One strength of the Transmission Control Protocol (TCP) [RFC0793] lies in its ability to adjust its sending rate according to the perceived congestion in the network [RFC5681]. In the absence of explicit notification of congestion from the network, TCP uses segment loss as an indication of congestion (i.e., assuming queue overflow). A TCP receiver sends cumulative acknowledgments (ACKs) indicating the next sequence number expected from the sender for arriving segments [RFC0793]. When segments arrive out of order, duplicate ACKs are generated. As specified in [RFC5681], a TCP sender uses the arrival of three duplicate ACKs as an indication of segment loss. The TCP sender retransmits the segment assumed lost and reduces the sending rate, based on the assumption that the loss was caused by resource contention on the path. The TCP sender does not assume loss on the first or second duplicate ACK, but waits for three duplicate ACKs to account for minor packet reordering. However, the use of this constant threshold of duplicate ACKs leads to performance degradation if the extent of the packet reordering in the network increases [RFC4653].

Whenever interoperability with the TCP congestion control and loss recovery standard [RFC5681] is a prerequisite, increasing the duplicate acknowledgment threshold (DupThresh) is the method of choice to a priori prevent any negative impact - in particular, a spurious Fast Retransmit and Fast Recovery phase - that packet reordering has on TCP. However, this procedure also delays a Fast Retransmit by increasing the DupThresh, and therefore has costs and risks, too. According to [ZKFP03], these are: (1) a delayed response to congestion in the network, (2) a potential expiration of the retransmission timer, and (3) a significant increase in the end-to-end delay for lost segments.

In the current TCP standard, congestion control and loss recovery are tightly coupled: when the oldest outstanding segment is declared lost, a retransmission is triggered, and the sending rate is reduced on the assumption that the loss is due to resource contention [RFC5681]. Therefore, any change to DupThresh causes not only a change to the loss recovery, but also to the congestion control response. TCP-NCR [RFC4653] addresses this problem by defining two extensions to TCP's Limited Transmit [RFC3042] scheme: Careful and Aggressive Extended Limited Transmit.

The first variant of the two, Careful Limited Transmit, sends one previously unsent segment in response to duplicate acknowledgments for every two segments that are known to have left the network. This effectively halves the sending rate, since normal TCP operation sends one new segment for every segment that has left the network. Further, the halving starts immediately and is not delayed until a retransmission is triggered. In the case of packet reordering (i.e., not segment loss), TCP-NCR restores the congestion control state to its previous state after the event.

The second variant, Aggressive Limited Transmit, transmits one previously unsent data segment in response to duplicate acknowledgments for every segment known to have left the network. With this variant, while waiting to disambiguate the loss from a reordering event, ACK-clocked transmission continues at roughly the same rate as before the event started. Retransmission and the sending rate reduction happen per [RFC5681] [RFC6675], albeit after a delay caused by the increased DupThresh. Although this approach delays legitimate rate reductions (possibly slightly, and temporarily aggravating overall congestion on the network), the scheme has the advantage of not reducing the transmission rate in the face of packet reordering.

A basic requirement for preventing an avoidable expiration of the retransmission timer is to generally ensure that an increased DupThresh can potentially be reached in time so that Fast Retransmit is triggered and Fast Recovery is completed before the RTO expires. Simply increasing DupThresh before retransmitting a segment can make TCP brittle to packet or ACK loss, since such loss reduces the number of duplicate ACKs that will arrive at the sender from the receiver. For instance, if cwnd is 10 segments and one segment is lost, a DupThresh of 10 will never be met, because duplicate ACKs corresponding to at most 9 segments will arrive at the sender. To mitigate this issue, the TCP-NCR [RFC4653] modification makes two fundamental changes to the way [RFC5681] [RFC6675] currently operates.

First, as mentioned above, TCP-NCR [RFC4653] extends TCP’s Limited Transmit [RFC3042] scheme to allow for the sending of new data segment while the TCP sender stays in the ’disorder’ state and disambiguate loss and reordering. This new data serves to increase the likelihood that enough duplicate ACKs arrive at the sender to trigger loss recovery, if it is appropriate. Second, DupThresh is increased from the current fixed value of three [RFC5681] to a value indicating that approximately a congestion window’s worth of data has left the network. Since cwnd represents the amount of data a TCP sender can transmit in one round-trip time (RTT), this corresponds to

approximately the largest amount of time a TCP sender can wait before the costly retransmission timeout may be triggered.

Of vital importance is that TCP-NCR [RFC4653] holds DupThresh not constant, but dynamically adjusts it on each SACK to the current amount of outstanding data, which depends not only on the congestion window, but also on the receiver's advertised window. Thus, it is guaranteed that the outstanding data generates a sufficient number of duplicate ACKs for reaching DupThresh and a transition to the 'recovery' state. This is important in cases where there is no new data available to send.

Regarding the problem of packet reordering, TCP-NCR's [RFC4653] decision of waiting to receive notice that cwnd bytes have left the network before deciding whether the root cause is loss or reordering is essentially a trade-off between making the best decision regarding the cause of the duplicate ACKs and responsiveness, and represents a good compromise between avoiding spurious Fast Retransmits and avoiding unnecessary RTOs. On the other hand, if there is no visible packet reordering on the network path - which today is the rule and not the exception - or the delay caused by the reordering is very low, delaying Fast Retransmit is unnecessary in the case of congestion, and data is delivered to the application up to one RTT later. Especially for delay-sensitive applications, such as a terminal session over SSH, this is generally undesirable. By dynamically adapting DupThresh not only to the amount of outstanding data but also to the perceived packet reordering on the network path, this issue can be offset. This is the key idea behind the TCP-aNCR algorithm.

This document specifies a set of TCP modifications to provide an adaptive Non-Congestion Robustness (aNCR) mechanism for TCP. The TCP-aNCR modifications lend themselves to incremental deployment. Only the TCP implementation on the sender side requires modification. The changes themselves are modest. TCP-aNCR is built on top of the TCP Selective Acknowledgments Option [RFC2018] and the SACK-based loss recovery scheme given in [RFC6675] and represents an enhancement of the original TCP-NCR mechanism [RFC4653]. Currently, TCP-aNCR is an independent approach of making TCP more robust to packet reordering. It is not clear if upcoming versions of this draft TCP-aNCR will obsolete TCP-NCR or not.

It should be noted that the TCP-aNCR algorithm in this document could be easily adapted to the Stream Control Transmission Protocol (SCTP) [RFC2960], since SCTP uses congestion control algorithms similar to TCP (and thus has the same reordering robustness issues).

The remainder of this document is organized as follows. Section 3 provides a high-level description of the TCP-aNCR mechanism. Section 4 defines TCP-aNCR's requirements for an appropriate detection and quantification algorithm. Section 5 specifies the TCP-aNCR algorithm and Section 6 discusses each step of the algorithm in detail. Section 7 provides a discussion of several design decisions behind TCP-aNCR. Section 8 discusses interoperability issues related to introducing TCP-aNCR. Finally, related work is presented in Section 9 and security concerns in Section 11.

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 [RFC2119].

The reader is expected to be familiar with the TCP state variables described in [RFC0793] (SND.NXT), [RFC5681] (cwnd, rwnd, ssthresh, FlightSize, IW), [RFC6675] (pipe, DupThresh, SACK scoreboard), and [RFC6582] (recover). Further, the term 'acceptable acknowledgment' is used as defined in [RFC0793]. That is, an ACK that increases the connection's cumulative ACK point by acknowledging previously unacknowledged data. The term 'duplicate acknowledgment' is used as defined in [RFC6675], which is different from the definition of duplicate acknowledgment in [RFC5681].

This specification defines the four TCP sender states 'open', 'disorder', 'recovery', and 'loss' as follows. As long as no duplicate ACK is received and no segment is considered lost, the TCP sender is in the 'open' state. Upon the reception of the first consecutive duplicate ACK, TCP will enter the 'disorder' state. After receiving DupThresh duplicate ACKs, the TCP sender switches to the 'recovery' state and executes standard loss recovery procedures like Fast Retransmit and Fast Recovery [RFC5681]. Upon a retransmission timeout, the TCP sender enters the 'loss' state. The 'recovery' state can only be reached by a transition from the 'disorder' state, the 'loss' state can be reached from any other state.

The following specification depends on the standard TCP congestion control and loss recovery algorithms and the SACK-based loss recovery scheme given in [RFC5681], respectively [RFC6675]. The algorithm presents an enhancement of TCP-NCR [RFC4653]. The reader is assumed to be familiar with the algorithms specified in these documents.

3. Basic Concept

The general idea behind the TCP-aNCR algorithm is to extend the TCP-NCR algorithm [RFC4653], so that - based on an appropriate packet reordering detection and quantification algorithm (see Section 4) - TCP congestion control and loss recovery [RFC5681] is adaptively adjusted to the actual perceived packet reordering on the network path.

TCP-NCR [RFC4653] increases DupThresh from the current fixed value of three duplicate ACKs [RFC5681] to approximately until a congestion window of data has left the network. Since cwnd represents the amount of data a TCP sender can transmit in one RTT, the choice to trigger a retransmission only after a cwnd's worth of data is known to have left the network represents roughly the largest amount of time a TCP sender can wait before the RTO may be triggered. The approach chosen in TCP-aNCR is to take TCP-NCR's DupThresh as an upper bound for an adjustment of the DupThresh that is adaptive to the actual packet reordering on the network path.

Using TCP-NCR's DupThresh as an upper bound decouples the avoidance of spurious Fast Retransmits from the avoidance of unnecessary retransmission timeouts. Therefore, the adaptive adjustment of the DupThresh to current perceived packet reordering can be conducted without taking any retransmission timeout avoidance strategy into account. This independence allows TCP-aNCR to quickly respond to perceived packet reordering by setting its DupThresh so that it always corresponds to the minimum of the maximum possible (TCP-NCR's DupThresh) and the maximum measured reordering extent since the last RTO. The reordering extent used by TCP-aNCR is by itself not a static absolute reordering extent, but a relative reordering extent (see Section 4).

4. Appropriate Detection and Quantification Algorithms

If the TCP-aNCR algorithm is implemented at the TCP sender, it MUST be implemented together with an appropriate packet reordering detection and quantification algorithm that is specified in a standards track or experimental RFC.

Designers of reordering detection algorithms who want their algorithms to work together with the TCP-aNCR algorithm SHOULD reuse the variable 'ReorExtR' (relative reordering extent) with the semantics and defined values specified in [I-D.zimmermann-tcpm-reordering-detection]. A 'ReorExtR' given by the detection algorithm holds a value ranging from 0 to 1 which holds the new measured reordering sample as a fraction of the data in

flight. TCP-aNCR then saves this new fraction if it is greater than the current value.

5. The TCP-aNCR Algorithm

When both the Nagle algorithm [RFC0896] [RFC1122] and the TCP Selective Acknowledgment Option [RFC2018] are enabled for a connection, a TCP sender MAY employ the following TCP-aNCR algorithm to dynamically adapt TCP's congestion control and loss recovery [RFC5681] to the currently perceived packet reordering on the network path.

Without the Nagle algorithm, there is no straightforward way to accurately calculate the number of outstanding segments in the network (and, therefore, no good way to derive an appropriate DupThresh) without adding state to the TCP sender. A TCP connection that does not use the Nagle algorithm SHOULD NOT use TCP-aNCR. The adaptation of TCP-aNCR to an implementation that carefully tracks the sequence numbers transmitted in each segment is considered future work.

A necessary prerequisite for TCP-aNCR's adaptability is that a TCP sender has enabled an appropriate detection and quantification algorithm that complies with the requirements defined in Section 4. If such an algorithm is either non-existent or not used, the behavior of TCP-aNCR is completely analogous to the TCP-NCR algorithm as defined in [RFC4653]. If a TCP sender does implement TCP-aNCR, the implementation MUST follow the various specifications provided in Sections 5.1 to 5.7.

5.1. Initialization during Connection Establishment

After the completion of the TCP connection establishment, the following state constants and variables MUST be initialized in the TCP transmission control block for the given TCP connection:

(C.1) Depending on which variant of Extended Limited Transmit should be executed, the constant LT_F MUST initialized as follows. For Careful Extended Limited Transmit:

$$LT\_F = 2/3$$

For Aggressive Extended Limited Transmit:

$$LT\_F = 1/2$$

This constant reflects the fraction of outstanding data (including data sent during Extended Limited Transmit) that

must be SACKed before a retransmission is at the latest triggered.

(C.2) If TCP-aNCR should adaptively adjust the DupThresh to the current perceived packet reordering on the network path, then the variable 'ReorExtR', which stores the maximum relative reordering extent, MUST initialized as:

```
ReorExtR = 0
```

Otherwise the dynamically adaptation of TCP-aNCR SHOULD be disabled by setting

```
ReorExtR = -1
```

A relative reordering extent of 0 results in the standard DupThresh of three duplicate ACKs, as defined in [RFC5681]. A fixed relative reordering extent of -1 results in the TCP-NCR behavior from [RFC4653].

5.2. Initializing Extended Limited Transmit

If the SACK scoreboard is empty upon the receipt of a duplicate ACK (i.e., the TCP sender has received no SACK information from the receiver), a TCP sender MUST enter Extended Limited Transmit by initialize the following five state variables in the TCP Transmission Control Block:

(I.1) The TCP sender MUST save the current outstanding data:

```
FlightSizePrev = FlightSize
```

(I.2) The TCP sender MUST save the highest sequence number transmitted so far:

```
recover = SND.NXT - 1
```

Note: The state variable 'recover' from [RFC6582] can be reused, since NewReno TCP uses 'recover' at the initialization of a loss recovery procedure, whereas TCP-aNCR uses 'recover' *before* loss recovery.

(I.3) The TCP sender MUST initialize the variable 'skipped' that tracks the number of segments for which an ACK does not trigger a transmission during Careful Limited Transmit:

```
skipped = 0
```

During Aggressive Limited Transmit, 'skipped' is not used.

(I.4) The TCP sender MUST set DupThresh based on the current FlightSize:

```
DupThresh = max (LT_F * (FlightSize / SMSS), 3)
```

The lower bound of DupThresh = 3 is kept from [RFC5681] [RFC6675].

(I.5) If (ReorExtR != -1) holds, then the TCP sender MUST set DupThresh based on the relative reordering extent 'ReorExtR':

```
DupThresh =
     max (min (DupThresh,
               ReorExtR * (FlightSizePrev / SMSS)), 3)
```

In addition to the above steps, the incoming ACK MUST be processed with the (E) series of steps in Section 5.3.

5.3. Executing Extended Limited Transmit

On each ACK that a) arrives after TCP-aNCR has entered the Extended Limited Transmit phase (as outlined in Section 5.2) *and* b) carries new SACK information, *and* c) does *not* advance the cumulative ACK point, the TCP sender MUST use the following procedure.

(E.1) The TCP sender MUST update the SACK scoreboard and uses the SetPipe() procedure from [RFC6675] to set the 'pipe' variable (which represents the number of bytes still considered "in the network"). Note: the current value of DupThresh MUST be used by SetPipe() to produce an accurate assessment of the amount of data still considered in the network.

(E.2) The TCP sender MUST initialize the variable 'burst' that tracks the number of segments that can at most be sent per ACK to the size of the Initial Window (IW) [RFC5681]:

```
burst = IW
```

(E.3) If a) (cwnd - pipe - skipped >= 1 * SMSS) holds, *and* b) the receive window (rwnd) allows to send SMSS bytes of previously unsent data, *and* c) there are SMSS bytes of previously unsent data available for transmission, then the TCP sender MUST transmit one segment of SMSS bytes. Otherwise, the TCP sender MUST skip to step (E.7).

(E.4) The TCP sender MUST increment 'pipe' by SMSS bytes and MUST decrement 'burst' by SMSS bytes to reflect the newly transmitted segment:

```
pipe = pipe + SMSS
burst = burst - SMSS
```

(E.5) If Careful Limited Transmit is used, 'skipped' MUST be incremented by SMSS bytes to ensure that the next SMSS bytes of SACKed data processed do not trigger a Limited Transmit transmission.

```
skipped = skipped + SMSS
```

(E.6) If (burst > 0) holds, the TCP sender MUST return to step (E.3) to ensure that as many bytes as appropriate are transmitted. Otherwise, if more than IW bytes were SACKed by a single ACK, the TCP sender MUST skip to step (E.7). The additional amount of data becomes available again by the next received duplicate ACK and the re-execution of SetPipe().

(E.7) The TCP sender MUST save the maximum amount of data that is considered to have been in the network during the last RTT:

```
pipe_max = max (pipe, pipe_max)
```

(E.8) The TCP sender MUST set DupThresh based on the current FlightSize:

```
DupThresh = max (LT_F * (FlightSize / SMSS), 3)
```

The lower bound of DupThresh = 3 is kept from [RFC5681] [RFC6675].

(E.9) If (ReorExtR != -1) holds, then the TCP sender MUST set DupThresh based on the relative reordering extent 'ReorExtR':

```
DupThresh =
     max (min (DupThresh,
               ReorExtR * (FlightSizePrev / SMSS)), 3)
```

5.4. Terminating Extended Limited Transmit

On the receipt of a duplicate ACK that a) arrives after TCP-aNCR has entered the Extended Limited Transmit phase (as outlined in Section 5.2) *and* b) advances the cumulative ACK point, the TCP sender MUST use the following procedure.

The arrival of an acceptable ACK that advances the cumulative ACK point while in Extended Limited Transmit, but before loss recovery is triggered, signals that a series of duplicate ACKs was caused by reordering and not congestion. Therefore, Extended Limited Transmit will be either terminated or re-entered.

(T.1) If the received ACK extends not only the cumulative ACK point, but *also* carries new SACK information (i.e., the ACK is both an acceptable ACK and a duplicate ACK), the TCP sender MUST restart Extended Limited Transmit and MUST go to step (T.2). Otherwise, the TCP sender MUST terminate it and MUST skip to step (T.3).

(T.2) If the Cumulative Acknowledgment field of the received ACK covers more than 'recover' (i.e., SEG.ACK > recover), Extended Limited Transmit has transmitted one cwnd worth of data without any losses and the TCP sender MUST update the following state variables by

```
FlightSizePrev = pipe_max
pipe_max = 0
```

and MUST go to step (I.2) to re-start Extended Limited Transmit. Otherwise if (SEG.ACK <= recover) holds, the TCP sender MUST go to step (I.3). This ensures that in the event of a loss the cwnd reduction is based on a current value of FlightSizePrev.

The following steps are executed only if the received ACK does *not* carry SACK information. Extended Limited Transmit will be terminated.

(T.3) A TCP sender MUST set ssthresh to:

```
ssthresh = max (cwnd, ssthresh)
```

This step provides TCP-aNCR with a sense of "history". If the next step (T.4) reduces the congestion window, this step ensures that TCP-aNCR will slow-start back to the operating point that was in effect before Extended Limited Transmit.

(T.4) A TCP sender MUST reset cwnd to:

```
cwnd = FlightSize + SMSS
```

This step ensures that cwnd is not significantly larger than the amount of data outstanding, a situation that would cause a line rate burst.

(T.5) A TCP is now permitted to transmit previously unsent data as allowed by cwnd, FlightSize, application data availability, and the receiver's advertised window.

5.5. Entering Loss Recovery

The receipt of an ACK that results in deeming the oldest outstanding segment is lost via the algorithms in [RFC6675] terminates Extended Limited Transmit and initializes the loss recovery according to [RFC6675]. One slight change to either [RFC6675], or, if Proportional Rate Reduction (PRR) algorithm is used, to [RFC6937] MUST be made, however.

(Ret) If the PRR algorithm is used to calculate how many bytes should be sent in response to each ACK, the initalization of 'RecoverFS' in Section 3 of [RFC6937] MUST be changed to:

```
RecoverFS = FlightSizePrev
```

Otherwise, if the standard Fast Recovery algorithm is used, step (4.2) of [RFC6675] MUST be changed in Section 5 to:

```
ssthresh = cwnd = (FlightSizePrev / 2)
```

This change ensures that the congestion control modifications are made with respect to the amount of data in the network before FlightSize was increased by Extended Limited Transmit.

Once the algorithm in [RFC6675] takes over from Extended Limited Transmit, the DupThresh value MUST be held constant until the loss recovery phase terminates.

5.6. Reordering Extent

Whenever the additional detection and quantification algorithm (see Section 4) detects and quantifies a new reordering event, the TCP sender MUST update the state variable 'ReorExtR'.

(Ext) Let 'ReorExtR_New' the newly determined relative reordering extent:

```
ReorExtR = min (max (ReorExtR, ReorExtR_New), 1)
```

5.7. Retransmission Timeout

The expiration of the retransmission timer SHOULD be interpreted as an indication of a path characteristics change, and the TCP sender SHOULD reset DupThresh to the default value of three.

(RTO) If an RTO occurs and (ReorExtR != -1) (i.e. TCP-aNCR is used and not TCP-NCR), then a TCP sender SHOULD reset 'ReorExtR':

ReorExtR = 0

6. Protocol Steps in Detail

Upon the receipt of the first duplicate ACK in the 'open' state (the SACK scoreboard is empty), the TCP sender starts to execute TCP-aNCR by entering the 'disorder' state and the initialization of Extended Limited Transmit. First, the TCP sender saves the current amount of outstanding data as well as the highest sequence number transmitted so far (SND.NXT - 1) (steps (I.1) and (I.2)). In addition, if the TCP connection uses the careful variant of the Extended Careful Limited Transmit (step (C.1)), the 'skipped' variable, which tracks the number of segments for which an ACK does not trigger a transmission during Careful Limited Transmit, is initialized with zero (step (I.3)). The last step during the initialization is the determination of DupThresh. Depending on whether TCP-aNCR has been configured during the connection establishment to adaptively adjust to the currently perceived packet reordering on the path (step (C.2)), DupThresh is either determined exclusively based on the current FlightSize (as TCP-NCR [RFC4653] does) or, in addition, also based on the relative extent reordering (steps (I.4) and (I.5)).

Depending on which variant of Extended Limited Transmit should be executed, the constant LT_F must be set accordingly (step (C.1)). This constant reflects the fraction of outstanding data (including data sent during Extended Limited Transmit) that must be SACKed before a retransmission is triggered at the latest (which is the case when a DupThresh that is based on relative reordering extent is larger then TCP-NCR's DupThresh). Since Aggressive Limited Transmit sends a new segment for every segment known to have left the network, a total of approximately cwnd segments will be sent, and therefore ideally a total of approximately 2*cwnd segments will be outstanding when a retransmission is finally triggered. DupThresh is then set to LT_F = 1/2 of 2*cwnd (or about 1 RTT's worth of data) (see step (I.4)). The factor is different for Careful Limited Transmit, because the sender only transmits one new segment for every two segments that are SACKed and therefore will ideally have a total of maximum of 1.5*cwnd segments outstanding when the retransmission is triggered. Hence, the required threshold is LT_F=2/3 of 1.5*cwnd to delay the retransmission by roughly 1 RTT.

For each duplicate ACK received in the 'disorder' state, which is not an acceptable ACK, i.e., it carries new SACK information, but does not advance the cumulative ACK point, Extended Limited Transmit is executed. First, the SACK scoreboard is updated and based on the

current value of DupThresh, the amount of outstanding data (step (E.1)). Furthermore, the state variable 'burst' that indicates the number of segments that can be sent at most for of each received ACK is initialized to the size of the initial window [RFC6928] (step E.2)). If more than IW bytes were SACKed by a single ACK, the additional amount of data becomes available again by the next received duplicate ACK and the re-execution of SetPipe() (step (E.1)).

Next, if new data is available for transmission and both the congestion window and the receiver window allow to send SMSS bytes of previously unsent data, a segment of SMSS bytes is sent (step (E.3)). Subsequently, the corresponding state variables 'pipe', 'burst' and - optionally - 'skipped' are updated (steps (E.4) and (E.5)). If, due to the current size of the congestion and receiver windows (step (E.2)), due to the current value of 'burst' (step (E.5)), no further segment may be sent, the processing of the ACK is terminated. Provided that the amount of data that is currently considered to be in the network is greater than the previously stored one, this new value is stored for later use (step (E.7)). Finally, to take into account the new data sent, DupThresh is updated (steps (E.6) and (E.7)).

The arrival of an acceptable ACK in the 'disorder' state that advances the cumulative ACK point during Extended Limited Transmit signals that a series of duplicate ACKs was caused by reordering and not congestion. Therefore, the receipt of an acceptable ACK that does not carry any SACK information terminates Extended Limited Transmit (step (T.1)). The slow start threshold is set to the maximum of its current value and the current value of cwnd (step (T.3)). Cwnd itself is set to the current value of FlightSize plus one segment (step (T.4)). As a result, the congestion window is not significantly larger than the current amount of outstanding data, so that a burst of data is effectively prevented. If new data is available for transmission and both the new values of cwnd and rwnd allow to send SMSS bytes of previously unsent data, a segment is send (step (T.5)).

On the other hand, if the received ACK acknowledges new data not only cumulatively but also selectively - the ACK carries new SACK information - Extended Limited Transmit is not terminated but re-entered (step (T.1)). If the Cumulative Acknowledgment field of the received ACK covers more than 'recover', one cwnd worth of data has been transmitted during Extended Limited Transmit without any packet loss. Therefore, FlightSizePrev, the amount of outstanding data saved at the beginning of Extended Limited Transmit (step (I.1)), is considered outdated (step (T.2)). This step ensures that in the event of packet loss, the reduction of the cwnd is based on an up-to-

date value, which reflects the number of bytes outstanding in the network (see Section 7). Finally, regardless of whether or not 'recover' is covered, Extended Limited Transmit is re-entered.

The second case that leads to a termination of Extended Limited Transmit is the receipt of an ACK that signals via the algorithm in [RFC6675] that the oldest outstanding segment is considered lost. If either DupThresh or more duplicate ACKs are received, or the oldest outstanding segment is deemed lost via the function IsLost() of [RFC6675], Extended Limited Transmit is terminated and SACK-based loss recovery is entered [RFC6675]. Once the algorithm in [RFC6675] takes over from Extended Limited Transmit, the DupThresh value MUST be held constant until loss recovery is terminated. The process of loss recovery itself is not changed by TCP-aNCR. The only exception is a slight change to either RFC 6675 [RFC6675] or RFC 6937 [RFC6937], depending on whether the PRR algorithm or the traditional Fast Recovery algorithm is used during loss recovery. This change ensures that the adjustment made by the congestion control - the cwnd reducation - is made with respect to the initial amount of outstanding data while Limited Transmit Extended is executed (step (Ret)). The use of FlightSize at this point would no longer be valid since the amount of outstanding data may double by executing Extended Limited Transmit.

7. Discussion of TCP-aNCR

The specification of TCP-aNCR represents an incremental update of RFC 4653 [RFC4653]. All changes made by TCP-aNCR can be divided into two categories. On one hand, they implement TCP-aNCR's ability to dynamically adapted TCP congestion control and loss recovery [RFC5681] to the currently perceived packet reordering on the network path. These include the use of a variable DupThresh and the use of a relative reordering extent. On the other hand, the changes that basically correct weaknesses of the original TCP-NCR algorithm and which are independent of TCP-aNCR adaptability. These include packet reordering during slow start, the prevention of bursts, and the persistent receipt of SACKs.

7.1. Variable Duplicate Acknowledgment Threshold

The central point of the TCP-aNCR algorithm is the usage of a DupThresh that is adaptable to the perceived packet reordering on the network path. Based on the actual amount of outstanding data, TCP-NCR's DupThresh represents roughly the largest amount of time a Fast Retransmit can safely be delayed before a costly retransmission timeout may be triggered. Therefore, to avoid an RTO, TCP-aNCR's reordering-aware DupThresh is an upper bound of the one calculated in TCP-NCR (steps (I.5) and (E.9)). This decouples the avoidance of

spurious Fast Retransmits from the avoidance of RTOs. It allows TCP-aNCR to react fast and efficiently to packet reordering. The DupThresh always corresponds to the minimum of the largest possible and largest detected reordering. With constant packet reordering in terms of the rate and delay, TCP-aNCR gives a DupThresh based on the relative reordering extent with an optimal delay for every bandwidth-delay-product. If TCP-aNCR should not adaptively adjust the DupThresh to the current perceived packet reordering on the network path (because for example an appropriate detection and quantification algorithm is not implemented), the dynamically adaptation of TCP-aNCR can be disabled, so that TCP-aNCR behaves like TCP-NCR [RFC4653].

7.2. Relative Reordering Extent

Whenever a new reordering event is detected and presented to TCP-aNCR in the form of a relative reordering extend 'ReorExtR', TCP-aNCR saves and uses the new 'ReorExtR' if it is larger than the old one (step (EXT)). The upper bound of 1 assures that no excessively large value is used. A 'ReorExtR' larger than one means that more than FlightSize bytes would have been received out-of-order before the reordered segment is received. The delay caused by the reordering is thus longer than the RTT of the TCP connection. Since the RTT is roughly the time a Fast Retransmit can safely be delayed before the retransmission has to be to avoid an RTO, a maximum 'ReorExtR' of one seems to be a suitable value.

The expiration of the retransmission timer is interpreted by TCP-aNCR as an indication of a change in path characteristics, hence, the saved 'ReorExtR' is assumed to be outdated and will be invalidated (step (RTO)). As a consequence, the relative reordering extent 'ReorExtR' increases monotonically between two successive retransmission timeouts and corresponds to the maximum measured reordering extent since the last RTO. Other approaches would be an exponentially-weighted moving average (EWMA) or a histogram of the last n reordering extents. The main drawback of an EWMA is however that on average half of the detected reordering events would be larger than the saved reordering extend. Thus, only half of the spurious retransmits could be avoided. Applying an histogram could largely avoid the disadvantages of an EWMA, however, it would result in a not acceptable increase in memory usage.

In combination with the invalidation after an RTO, the advantage of using maximum is the low complexity as well as its fast convergence to the actual maximum reordering on the network path. As a result, the negative impact that packet reordering has on TCP's congestion control and loss recovery can be avoided. A disadvantage of using a maximum is that if the delay caused by the reordering decreases over the lifetime of the TCP connection, a Fast Retransmit is

unnecessarily long delayed. Nevertheless, since the negative impact reordering has on TCP's congestion control and loss recovery is more substantial than the disadvantage of a longer delay, a decrease of the ReorExtR between RTOs is considered inappropriate.

7.3. Reordering during Slow Start

The arrival of an acceptable ACK during Extended Limited Transmit signals that previously received duplicate ACKs are the result of packet reordering and not congestion, so that Extended Limited Transmit is completed accordingly. Upon the termination of Extended Limited Transmit, and especially when using the Careful variant, TCP-NCR (as well as TCP-aNCR) may be in a situation where the entire cwnd is not being utilized. Therefore, to mitigate a potential burst of segments, in step (T.2) TCP-NCR sets the slow start threshold to the FlightSize that was saved at the beginning of Extended Limited Transmit [RFC4653]. This step should ensure that TCP-NCR slow starts back to the operating point in use before Extended Limited Transmit.

Unfortunately, the assignment in step (T.2) is only correct if the TCP sender already was in congestion avoidance at the time Extended Limited Transmit was entered. Otherwise, if the TCP sender was instead in slow start, the value of ssthresh is greater than the saved FlightSize so that slow start prematurely concludes. This behavior can leave much of the network resources idle, and a long time may needed in order to use the full capacity. To mitigate this issue, TCP-aNCR sets the slow start threshold to the maximum of its current value and the current cwnd (step (T.3)). This continues slow start after a reordering event happening during slow start.

7.4. Preventing Bursts

In cases where a new single SACK covers more than one segment - this can happen either due to packet loss or packet reordering on the ACK path - TCP-NCR [RFC4653] sends an undesirable burst of data. TCP-aNCR solves this problem by limiting the burst size - the maximum of data that can send in response to a single SACK - to the Initial Window [RFC5681] while executing Extended Limited Transmit (steps (E.2), (E.4), and (E.6)). Since IW represents the amount of data that a TCP sender is able to send into the network safely without knowing its characteristics, it is a reasonable value for the burst size, too. If more than IW bytes were SACKed by a single ACK, the additional amount of data becomes available again by the next received duplicate ACK. Thus, the transmission of new segments is spread over the next received ACKs, so that micro bursts - a characteristic of packet reordering in the reverse path - are largely compensated.

Another situation that causes undesired bursts of segments with TCP-NCR is the receipt of an acceptable ACK during Careful Extended Limited Transmit. If multiple segments from a single window of data are delayed by packet reordering, typically the first acceptable ACK after entering the 'disorder' state acknowledges data not only cumulatively but also selectively. Hence, Extended Limited Transmit is not terminated but re-started. If the segments are delayed by the reordering for almost one RTT, then the amount of outstanding data in the network ('pipe') is approximately half the amount of data saved at the beginning of Extended Limited Transmit (FlightSizePrev). If the sequence numbers of the delayed segments are close to each other in the sequence number space, the acceptable ACK acknowledges only a small amount of data, so that FlightSize is still large. As a result, TCP-NCR sets the cwnd to FlightSizePrev in step (T.1). Since 'pipe' is only half of FlightSizePrev due to Careful Extended Limited Transmit, TCP-NCR sends a burst of almost half a cwnd worth of data in the subsequent step (T.3).

Note: Even in the case the sequence numbers of the delayed segments are not close to each other in the sequence number space and cwnd is set in step (T.1) to FlightSize + SMSS, a burst of data will emerge due to re-entering Extended Limited Transmit, because TCP-NCR sets 'skipped' to zero in step (I.2) and uses FlightSizePrev in step (E.2).

TCP-aNCR prevents such a burst by making a clear differentiation between terminating Extended Limited Transmit and a restarting Extended Limited Transmit (step T.1). Only the first case causes the congestion window to be set to the current FlightSize plus one segment. In the latter case, when re-entering Extended Limited Transmit, the congestion window is not adjusted and the original (T.1) of the TCP-NCR specification is omitted. The transmission of new data is then only performed after re-entering Extended Limited Transmit in step (E.2) of the TCP-aNCR specification, where the actual burst mitigation takes place.

7.5. Persistent receiving of Selective Acknowledgments

In some inconvenient cases it could happen that a TCP sender persistently receives SACK information due to reordering on the network path, e.g., if the segments are often and/or lengthy delayed by the packet reordering. With TCP-NCR, the persistent reception of SACKs causes Extended Limited Transmit to be entered with the first received duplicate ACK but never to be terminated if no packet loss occurs - for every received ACK, TCP-NCR either follows steps (E.1) to (E.6) or steps (T.1) to (T.4). In particular, TCP-NCR executes a) for every acceptable ACK step (T.4) and b) at any time step (I.1)

again. Hence, the amount of outstanding data saved at the beginning of Extended Limited Transmit, FlightSizePrev, is never updated.

An emerging problem in this context is that during Extended Limited Transmit TCP-NCR determines the transmission of new segments in step (E.2) solely on the basis of FlightSizePrev, so that an interim increase of the cwnd is not considered (according to [RFC5681], the congestion window is increased for every received acceptable ACK that advances the cumulative ACK point, no matter if it carries SACK information or not). As a result, TCP-NCR can only very slowly determine the available capacity of the communication path.

TCP-aNCR addresses this problem by limiting the amount of data that is allowed to be sent into the network during Extended Limited Transmit not on the basis of FlightSizePrev, but on the size of the congestion window. The equation in step E.3 of the TCP-aNCR specification is therefore equal to the one used in [RFC6675] (except for the 'skipped' variable). If an acceptable ACK is received during the execution of Extended Limited Transmit, re-entering Extended Limited Transmit makes any increase in cwnd immediately available. Hence, even in the case when persistently receiving SACKs, the available capacity of the communication path can be determined quickly.

Another problem resulting from persistently receiving SACKs, and which is related to the increase in cwnd in response to received acceptable ACKs, is the reduction of cwnd due to a packet loss. When a packet is considered lost, the congestion control adjustment is done with respect to the amount of outstanding data at the beginning of Extended Limited Transmit, FlightSizePrev (step (Ret)). As in the previous case, an increase in cwnd is again not taken into account. A simple solution to the problem would be to perform the window reduction not on the basis of FlightSizePrev but analogous to step (E.2) based on the current size of cwnd.

A problem with this solution is that cwnd can potentially be increased, although the TCP connection is limited by the application and not by cwnd. Although [RFC2861] specifies that an increase of cwnd is only applicable if cwnd is fully utilized, this behavior is not specified by any standards track document. But even this conservative increase behavior is guaranteed to not be conservative enough. If, from a single window of data, both segments are delayed but also lost, cwnd would first be increased in response to each received acceptable ACKs, while subsequently reduced due to the lost segments, which would not result in a halving of the cwnd any more.

The solution proposed by TCP-aNCR reuses the state variable 'recover' from [RFC6582] and adapts the approach taken by NewReno TCP and SACK

TCP to detect, with help of the state variable, the end of one loss recovery phase properly, allowing to recover multiple losses from a single window of data efficiently. Therefore, by entering the 'disorder' state and the starting Extended Limited Transmit, TCP-aNCR saves the highest sequence number sent so far in 'recover'. If a received acceptable ACK covers more than 'recover', one cwnd's worth of data has been transmitted during Extended Limited Transmit without any packet loss. Hence, FlightSizePrev can be updated by 'pipe_max', which reflects the maximum amount of data that is considered to have been in the network during the last RTT. This update takes an interim increase in cwnd into account, so that in case of packet loss, the reduction in cwnd can be based on the current value of FlightSizePrev.

8. Interoperability Issues

TCP-aNCR requires that both the TCP Selective Acknowledgment Option [RFC2018] as well as a SACK-based loss recovery scheme compatible to one given in [RFC6675] are used by the TCP sender. Hence, compatibility to both specifications is REQUIRED.

8.1. Early Retransmit

The specification of TCP-aNCR in this document and the Early Retransmit algorithm specified in [RFC5827] define orthogonal methods to modify DupThresh. Early Retransmit allows the TCP sender to reduce the number of duplicate ACKs required to trigger a Fast Retransmit below the standard DupThresh of three, if FlightSize is less than 4*SMSS and no new segment can be sent. In contrast, TCP-aNCR allows, starting from the minimum of three duplicate ACKs, to increase the DupThresh beyond the standard of three duplicate ACKs to make TCP more robust to packet reordering, if the amount of outstanding data is sufficient to reach the increased DupThresh to trigger Fast Retransmit and Fast Recovery.

8.2. Congestion Window Validation

The increase of the congestion window during application-limited periods can lead to an invalidation of the congestion window, in that it no longer reflects current information about the state of the network, if the congestion window might never have been fully utilized during the last RTT. According to [RFC2861], the congestion window should, first, only be increased during slow-start or congestion avoidance if the cwnd has been fully utilized by the TCP sender and, second, gradually be reduced during each RTT in which the cwnd was not fully used.

A problem that arises in this context is that during Careful Extended Limited Transmit, cwnd is not fully utilized due to the variable 'skipped' (see step (E.3)), so that - strictly following [RFC2861] - the congestion window should not be increased upon the receipt of an acceptable ACK. A trivial solution of this problem is to include the variable 'skipped' in the calculation of [RFC2861] to determine whether the congestion window is fully utilized or not.

8.3. Reactive Response to Packet Reordering

As a proactive scheme with the aim to a priori prevent the negative impact that packet reordering has on TCP, TCP-aNCR can conceptually be combined with any reactive response to packet reordering, which attempts to mitigate the negative effects of reordering a posteriori. This is because the modifications of TCP-aNCR to the standard TCP congestion control and loss recovery [RFC6675] are implemented in the 'disorder' state and are performed by the TCP sender before it enters loss recovery, while reactive responses to packet reordering operate generally after entering loss recovery, by undoing the unnecessarily changes to the congestion control state.

If unnecessary changes to the congestion control state are undone after loss recovery, which is typically the case if a spurious Fast Retransmit is detected based on the DSACK option [RFC3708][RFC4015], since first ACK carrying a DSACK option usually arrives at a TCP sender only after loss recovery has already terminated, it might happen that the restoring of the original value of the congestion window is done at a time at which the TCP sender is already back in again in the 'disorder' state and executing Extended Limited Transmit. While this is basically compatible with the TCP-aNCR specification - the undo simply represents an increase of the congestion window - however, some care must be taken that the combination of the algorithms does not lead to unwanted behavior.

8.4. Buffer Auto-Tuning

Although all modifications of the TCP-aNCR algorithm are implemented in the TCP sender, the receiver also potentially has a part to play. If some segments from a single window of data are delayed by the packet reordering in the network, all segments that are received in out-of-order have to be queued in the receive buffer until the holes in sequence number space have been closed and the data can be delivered to the receiving application. In the worst case, which occurs if the TCP sender uses Aggressive Limited Transmit and the reordering delay is close to the RTT, TCP-aNCR increases the receiver's buffering requirement by up to an extra cwnd. Therefore, to maximize the benefits from TCP-aNCR, receivers should advertise a large window - ideally by using buffer auto-tuning algorithms - to

absorb the extra out-of-order data. In the case that the additional buffer requirements are not met, the use of the above algorithm takes into account the reduced advertised window - with a corresponding loss in robustness to packet reordering.

9. Related Work

Over the past few years, several solutions have been proposed to improve the performance of TCP in the face of packet reordering. These schemes generally fall into one of two categories (with some overlap): mechanisms that try to prevent spurious retransmits from happening (proactive schemes) and mechanisms that try to detect spurious retransmits and undo the needless congestion control state changes that have been taken (reactive schemes).

[I-D.blanton-tcp-reordering], [ZKFP03] and [LM05] attempt to prevent packet reordering from triggering spurious retransmits by using various algorithms to approximate the DupThresh required to disambiguate loss and reordering over a given network path at a given time. This basic principle is also used in TCP-aNCR. While [I-D.blanton-tcp-reordering] describes four basic approaches on how to increase the DupThresh and discusses pros and cons of these approaches, presents [ZKFP03] a relatively complex algorithm that saves the reordering extents in a histogram and calculates the DupThresh in a way that a certain percentage of samples is smaller then the DupThresh. [LM05] uses an EWMA for the same purpose. Both algorithms do not prevent all the spurious retransmissions by design.

In contrast to the above mentioned algorithms Linux [Linux] implements a proactive scheme by setting the DupThresh to the highest detected reordering and resets only upon an RTO. To avoid a costly retransmission timeout due to the increased DupThresh Linux implements first an extension of the Limited Transmit algorithm, second limits the DupThresh to an upper bound of 127 duplicate ACKs, and third prematurely enters loss recovery if too few segments are in-flight to reach the DupThresh and no additional segments can send. Especially the last change is commendable since, besides TCP-NCR, none of the described algorithms in this section mention a similar concern.

[BHLLO06] and [BSRV04] presents proactive schemes based on timers by which the DupThresh is ignored altogether. After the timer is expired TCP initialize the loss recovery. In [BSRV04] this timer has a length of one RTT and is started when the first duplicate ACK is received, whereas the approach taken in [BHLLO06] solely relies on timers to detect packet loss without taking into account any other congestion signals such as duplicate ACKs. It assigns each segment

send a timestamp and retransmits the segment if the corresponding timer fires.

TCP-NCR [RFC4653] tries to prevent spurious retransmits similar to [I-D.blanton-tcp-reordering] or [ZKFP03] as it delays a retransmission to disambiguate loss and reordering. However, TCP-NCR takes a simplified approach by simply delay a retransmission by an amount based on the current cwnd (in comparison to standard TCP), while the other schemes use relatively complex algorithms in an attempt to derive a more precise value for DupThresh that depends on the current patterns of packet reordering. Many of the features offered by TCP-NCR have been taken into account while designing TCP-aNCR.

Besides the proactive schemes, several other schemes have been developed to detect and mitigate needless retransmissions after the fact. The Eifel detection algorithm [RFC3522], the detection based on DSACKs [RFC3708], and F-RTO scheme [RFC5682] represent approaches to detect spurious retransmissions, while the Eifel response algorithm [RFC4015], [I-D.blanton-tcp-reordering], and Linux [Linux] present respectively implement algorithms to mitigate the changes these events made to the congestion control state. As discussed in Section 8.3 TCP-aNCR could be used in conjunction with these algorithms, with TCP-aNCR attempting to prevent spurious retransmits and some other scheme kicking in if the prevention failed.

10. IANA Considerations

This memo includes no request to IANA.

11. Security Considerations

By taking dedicated actions so that the perceived packet reordering in the network is either underestimating or overestimating by the use of an relative and absolute reordering, an attacker or misbehaving TCP receiver has in regards to TCP’s congestion control two options to bias a TCP-aNCR sender. An underestimation of the present packet reordering in the network occursi, if for example, a misbehaving TCP receiver already acknowledges segments while they are actually still in-flight, causing holes premature are closed in the sequence number space of the SACK scoreboard. With regard to TCP-aNCR the result of an underestimated packet reordering is a too small DupThresh, resulting in a premature loss recovery execution. In context of TCP’s congestion control the effects of such attacks are limited since the lower bound of TCP-aNCR’s DupThresh is the default value of three duplicate ACKs [RFC5681], so that in worst case TCP-aNCR behaves equal to TCP SACK [RFC6675].

In contrast to an underestimation, an overestimation of the packet reordering in the network occurs, if for example, a misbehaving TCP receiver still further send SACKs for subsequent segments before it sends an acceptable ACK for the actually already received delayed segment, so that the hole in the sequence number space of the SACK scoreboard is later closed. In the context of TCP-aNCR the result of such an overestimation is a too large DupThresh, so that in the case of a packet loss TCP's loss recovery is executed later than necessary. Similar to the previous case, the effects of delayed entry into the loss recovery are limited because on the one hand TCP-NCR's DupThresh is used as an upper bound for TCP-aNCR's variable DupThresh so that the entrance to the loss recovery and the adaptation of the congestion window may be delayed at most one RTT. On the other hand, such a limited delay of the congestion control adjustment has even in the worst case only a limited impact on the performance of TCP connection and has generally been regarded as safe for use on the Internet [BBFS01].

12. Acknowledgments

The authors would like to thank Daniel Slot for his TCP-NCR implementation in Linux. We also thank the flowgrind [Flowgrind] authors and contributors for here performance measurement tool, which give us a powerful tool to analyze TCP's congestion control and loss recovery behavior in detail.

13. References

13.1. Normative References

[I-D.zimmermann-tcpm-reordering-detection]
Zimmermann, A., Schulte, L., Wolff, C., and A. Hannemann, "Detection and Quantification of Packet Reordering with TCP", draft-zimmermann-tcpm-reordering-detection-01 (work in progress), November 2013.

[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.

[RFC3042] Allman, M., Balakrishnan, H., and S. Floyd, "Enhancing TCP's Loss Recovery Using Limited Transmit", RFC 3042, January 2001.

[RFC4653] Bhandarkar, S., Reddy, A., Allman, M., and E. Blanton, "Improving the Robustness of TCP to Non-Congestion Events", RFC 4653, August 2006.

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

[RFC6582] Henderson, T., Floyd, S., Gurtov, A., and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 6582, April 2012.

[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.

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

[RFC6937] Mathis, M., Dukkipati, N., and Y. Cheng, "Proportional Rate Reduction for TCP", RFC 6937, May 2013.

13.2. Informative References

[BBFS01] Bansal, D., Balakrishnan, H., Floyd, S., and S. Shenker, "Dynamic Behavior of Slowly Responsive Congestion Control Algorithms", Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'01) pp. 263-274, September 2001.

[BHLLO06] Bohacek, S., Hespanha, J., Lee, J., Lim, C., and K. Obraczka, "A New TCP for Persistent Packet Reordering", IEEE/ACM Transactions on Networking vol. 2, no. 14, pp. 369-382, April 2006.

[BSRV04] Bhandarkar, S., Sadry, N., Reddy, A., and N. Vaidya, "TCP-DCR: A Novel Protocol for Tolerating Wireless Channel Errors", IEEE Transactions on Mobile Computing vol. 4, no. 5., pp. 517-529, September 2005.

[Flowgrind] "Flowgrind Home Page", <http://www.flowgrind.net>.

[I-D.blanton-tcp-reordering] Blanton, E., Dimond, R., and M. Allman, "Practices for TCP Senders in the Face of Segment Reordering", draft-blanton-tcp-reordering-00 (work in progress), February 2003.

[LM05] Leung, C. and C. Ma, "Enhancing TCP Performance to Persistent Packet Reordering", KICS Journal of Communications and Networks vol. 7, no. 3, pp. 385-393, September 2005.

[Linux] "The Linux Project", <http://www.kernel.org>.

[RFC0896] Nagle, J., "Congestion control in IP/TCP internetworks", RFC 896, January 1984.

[RFC1122] Braden, R., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, October 1989.

[RFC2861] Handley, M., Padhye, J., and S. Floyd, "TCP Congestion Window Validation", RFC 2861, June 2000.

[RFC2960] Stewart, R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang, L., and V. Paxson, "Stream Control Transmission Protocol", RFC 2960, October 2000.

[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.

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

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

[ZKFP03] Zhang, M., Karp, B., Floyd, S., and L. Peterson, "RR-TCP: A Reordering-Robust TCP with DSACK", Proceedings of the 11th IEEE International Conference on Network Protocols (ICNP’03) pp. 95-106, November 2003.

Appendix A. Changes from previous versions of the draft

This appendix should be removed by the RFC Editor before publishing this document as an RFC.

A.1. Changes from draft-zimmermann-tcpm-reordering-reaction-01

- Specify interaction between TCP-aNCR and PRR.
- Fix typo in DupThresh calculation (steps I.5 and E.9).

A.2. Changes from draft-zimmermann-tcpm-reordering-reaction-00

- Improved the wording throughout the document.
- Replaced and updated some references.

Authors' Addresses

Alexander Zimmermann
NetApp, Inc.
Sonnenallee 1
Kirchheim 85551
Germany

Phone: +49 89 900594712
Email: alexander.zimmermann@netapp.com

Lennart Schulte
Aalto University
Otakaari 5 A
Espoo 02150
Finland

Phone: +358 50 4355233
Email: lennart.schulte@aalto.fi

Carsten Wolff
credativ GmbH
Hohenzollernstrasse 133
Moenchengladbach 41061
Germany

Phone: +49 2161 4643 182
Email: carsten.wolff@credativ.de

Arnd Hannemann
credativ GmbH
Hohenzollernstrasse 133
Moenchengladbach 41061
Germany

Phone: +49 2161 4643 134
Email: arnd.hannemann@credativ.de