TCP Maintenance and Minor Extensions (tcpm)
Internet-Draft
Updates: 793 (if approved)
Intended status: Experimental
Expires: September 10, 2015

B. Briscoe
BT

March 09, 2015

Inner Space for all TCP Options (Kitchen Sink Draft - to be Split Up) draft-briscoe-tcpm-inspace-mode-tcpbis-00

Abstract

This document describes an experimental redesign of TCP's extensibility mechanism. It aims to traverse most known middleboxes including connection splitters, by making it possible to tunnel all TCP options within the TCP Data. It provides a choice between in-order and out-of-order delivery for TCP options. In-order delivery is a useful new facility for options that control datastream processing. Out-of-order delivery has been the norm for TCP options until now, and is necessary for options involved with acknowledging data, otherwise flow control can deadlock. TCP's original design limits TCP option space to 40B. In the new design there is no such arbitrary limit, other than the maximum size of a segment. The TCP client can immediately start to use the extra option space optimistically from the very first SYN segment, by using a dual handshake. The dual handshake is designed to prevent 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 is underway. In summary, the protocol should allow new TCP options to be introduced i) with minimal middlebox traversal problems; ii) with incremental deployment from legacy servers; iii) with zero handshaking delay iv) with a choice of in-order and out-of-order delivery v) without arbitrary limits on available space.

Status of This Memo

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

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

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any

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

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

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.

In 2011, [Honda11] tested a broad but small set of paths and found that there were few if any middlebox traversal problems over residential access networks, but the chance of a new option traversing other types of access was terrible. Cellular was especially bad (stripping options on 40% of paths for port 80 and 20% for other ports), but WiFi hotspots, enterprise, and university networks were close behind (typically, about 18% of paths blocked new extensions). 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 a middlebox resegments the datastream 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 for TCP options within the TCP Data is virtually unlimited--constrained only by the maximum segment size.

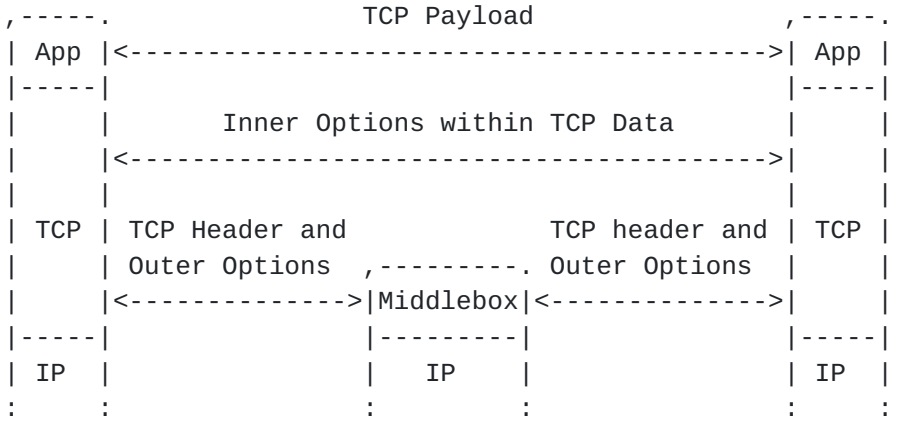


Figure 1: Encapsulation Approach

Tunnelling options within TCP Data raises two difficult questions: i) immediate (out-of-order) delivery of certain options and ii) bootstrapping the inner control channel.

Traditional TCP options [RFC0793] are delivered unreliably and out of order, because they are within the main header, outside the TCP sequence space. This document calls these 'Outer Options'. When TCP

options are placed within the TCP Data (Inner Options), it is easiest to include them within TCP's sequence space. Then TCP naturally delivers them reliably and in order without any extra machinery. However, in-order delivery is unacceptable for some options.

TCP options fall into three categories:

Segment-related (out-of-order): These have to be delivered to the receiver's TCP stack as soon as they are received (i.e. not necessarily in the order sent). They are generally concerned with transmission of each TCP segment, e.g. Timestamps, Selective ACKnowledgements (SACK), the Data ACK of Multipath TCP [RFC6824] and the message authentication code (MAC) of tcpcrypt [I-D.bittau-tcpinc-tcpcrypt].

Datastream-related (in-order): These would ideally be applied in the order that the sender inserted them into the datastream. They are generally concerned with controlling the transmission of the ordered datastream, e.g. the options of the TCP AO [RFC5925] that control data authentication or the suboptions of tcpcrypt that control data encryption [I-D.bittau-tcpinc-tcpcrypt]. At the time these were designed, TCP only provided Outer Options, so it was complex to apply TCP-AO options reliably and in order and similar complexity is being included in tcpcrypt;

Connection-related (order-agnostic): These are typically applied at the start of a connection which is also inherently the start of the first segment so the order of segment delivery is not a concern, e.g. TCP fast option [I-D.ietf-tcpm-fastopen], the sub-options of MPTCP [RFC6824] (except the Data ACK), and most of the TCP options that are in common usage;

The simplest ('default') variant of the Inner Space protocol [I-D.briscoe-tcpm-inner-space] delivers all Inner Options reliably and in order within the datastream.Therefore the default-mode Inner Space protocol can only support segment-related options as Outer Options. This is irritating because even though only a few options are segment-related, if just one kind of option cannot traverse a middlebox, it often prevents a whole set of other extensions from being used even though they would have no problem traversing the middlebox as Inner Options. For instance, one MPTCP option (the Data ACK) and one tcpcrypt option (the MAC) have to be delivered immediately (out of order), even though all the other MPTCP and tcpcrypt options can be delivered in order.

The present specification extends the default-mode Inner Space protocol to add out-of-order delivery of Inner Options. It can then support all TCP options as Inner Options. This offers the prospect

of completely circumventing middlebox problems and space problems for all TCP extensions.

The second difficult question addressed by the present specification is how to bootstrap the inner control channel--without any visible difference to the TCP wire protocol that would otherwise be unlikely to traverse many middleboxes. 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.

The Inner Space protocol will have to use whichever bootstrap approach is least bad, because they all involve compromises. For the present specification, the dual handshake has been chosen over the only other candidate currently in the running [I-D.touch-tcpm-tcp-syn-ext-opt], in which the client complements the SYN with an out-of-band (OOB) segment. In both approaches the client starts the connection with two segments. However, with the OOB approach the two segments will always be necessary, whereas the dual handshake is only a transition strategy that becomes unnecessary for each server as it is upgraded. Both approaches will need to be tested for middlebox traversal. It seems likely that many firewalls will block the OOB segment and it is also expected that some middleboxes will block the data in the SYN used for one of the dual handshakes.

In the dual handshake approach the client sends two SYNs; one 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 corrupts the application by passing it Inner Options. Otherwise, if the server does understand the new protocol, the client can abort the ordinary handshake, given it offers no extra option space. Either way, zero extra delay is added. 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.

Solving the five problems of i) option-space exhaustion; ii) middlebox traversal; iii) legacy server confusion; iv) a choice of in-order and out-of-order frame delivery; and v) handshake latency; does not come without cost:

- So that the Inner Space protocol is immune to option stripping, it avoids a conventional TCP option in the header. Instead it signals its presence using a magic number within the TCP Data of

the initial segment in each direction. 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. 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.

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

A number of extensions to TCP are in the process of definition and experimentation (TCPINC, MPTCP, etc). If a general-purpose middlebox traversal solution were available now, each new protocol design would not need complex machinery to detect and work round the byzantine range of middlebox behaviours. It would also make these extensions available to many more users.

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 C 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. Wider Implications

The implications of this work are more than 'just' a low latency incrementally deployable way to extend TCP option space:

End-to-middle signalling channel: Once endpoints have an end-to-end control channel within the TCP Data, they can use authentication or even encryption to stop middleboxes interfering with it. Then given middleboxes already interfere with Outer TCP Options, they can serve a new purpose as a channel for end-system TCP stacks to interact with middleboxes, but only if they choose to.

Multiplexed streams, compression, encryption (transport services): The Inner Space protocol has been designed generically, so that different delivery modes such as in-order and out-of-order delivery can be applied to different frames within the TCP Data. An additional mode could be added to extend out-of-order delivery to user-data, not just TCP control options. Then a single TCP connection could deliver data in multiple independent streams to minimise latency while one stream is blocked by a loss without the overhead of multiple connections. Inner Space is also structured so that data transformations such as compression or encryption can

easily be introduced and controlled by TCP options, as a generic facility available to any application layer protocol.

All these transport services (multiplexed streams, compression, encryption) are sought after by Web applications. However attempts to make them available in new transport protocols (e.g. SCTP) have proved impossible to deploy over the public Internet because too many middleboxes block new protocol identifiers. To work round this impasse, these transport services are being embedded within the application layer as part of the next generation of the HTTP protocol [I-D.ietf-httpbis-http2]. Inner Space has been designed so that these transport services would be straightforward to add in a structured way at the transport layer, using a new TCP mode. A separate document is planned to specify this mode. The present document focuses solely on TCP control options, which meets specific immediate needs. Nonetheless, the similarity is close enough to extrapolate that it will be straightforward to provide the transport services that Web applications need as well.

1.5. 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 5 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 defines the encoding that the Inner Space protocol uses for TCP Data. Eventually, this appendix is likely to be published separately because the encoding is more generally applicable. Appendix B defines an Inner TCP Option that provides a capability to switch the mode of a TCP connection, where the term 'mode' is a very general concept that might be used to change the ordering semantics of a connection, or switch off the Inner Space capability part way through a connection. Eventually this appendix is likely to be published separately due to its general applicability. Appendix C specifies optional extensions to the protocol that will need to be implemented experimentally to determine whether they are useful. And Appendix D discusses the merits of the chosen design against some of the optional extensions.

1.6. 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, in sequence with the data.

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 InSpace Option or any inner padding, 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, any inner padding and the InSpace Option defined in the present specification.

Pure ACK: A TCP acknowledgement with no TCP Data at all.

Impure ACK: A TCP acknowledgement with no TCP Payload or Suffix Options, but with at least an InSpace Option and possibly padding and Prefix Options.

Flow-Controlled ACK: A TCP acknowledgement containing at least TCP Payload and/or Suffix Options.

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; because it will always be legitimate to open an Ordinary Connection if extra option space or middlebox traversal 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

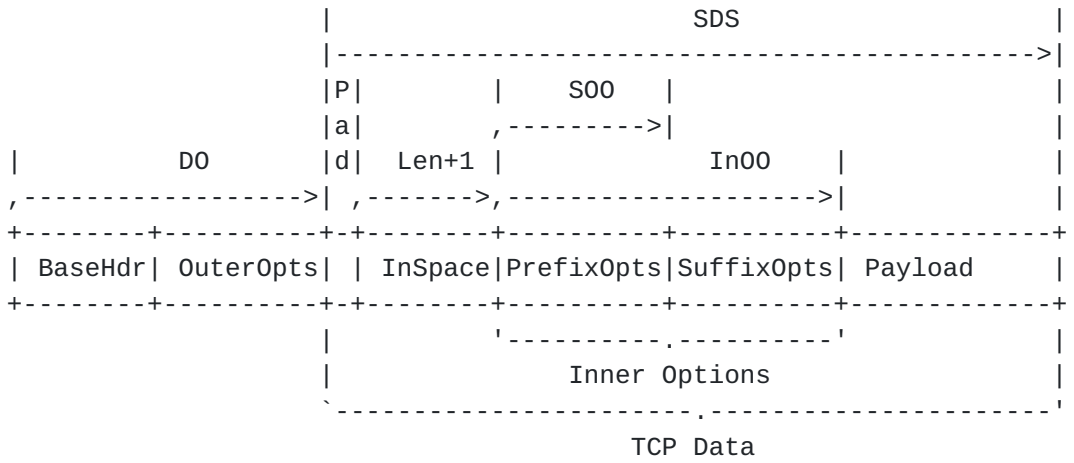
Once an Upgraded Connection has been successfully negotiated in the SYN, SYN/ACK exchange, either host can allocate any amount of the TCP Data space in any subsequent segment for extra TCP options. In fact, the sender has to use the upgraded segment structure in every subsequent segment of the connection that contains non-zero TCP Payload. The sender can use the upgraded structure in a segment carrying no TCP Payload, but it does not have to (see Section 2.3.1.5).

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 B. 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 modes 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.



All offsets are specified in 4-octet (32-bit) words, except SDS and Pad, which are 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.

Padding might have to be included at the start of the TCP Data to align the InSpace option on a 4-octet boundary from the start of the datastream (see Section 2.3.1.2).

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 Length (Len) of the InSpace option itself is read from a fixed location within the InSpace option.

The Sent Data Size (SDS) is also read from within the InSpace Option. If the datastream has been resegmented, it allows the receiver to know the size of the segment as it was when it was sent, even if the InSpace Options are no longer at the start of each segment (see Section 2.3).

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). The receiver processes PrefixOpts before OuterOpts, then SuffixOpts afterwards in order with the datastream. Full details of option processing are given in Section 2.3.

The first segment in each direction (i.e. the SYN or the SYN/ACK) is identifiable as upgraded by the presence of 6-octets of magic number 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 3.2.2). 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 then know the protocol that determines where subsequent InSpace options will be located.

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) or an abbreviated form in Figure 3c).

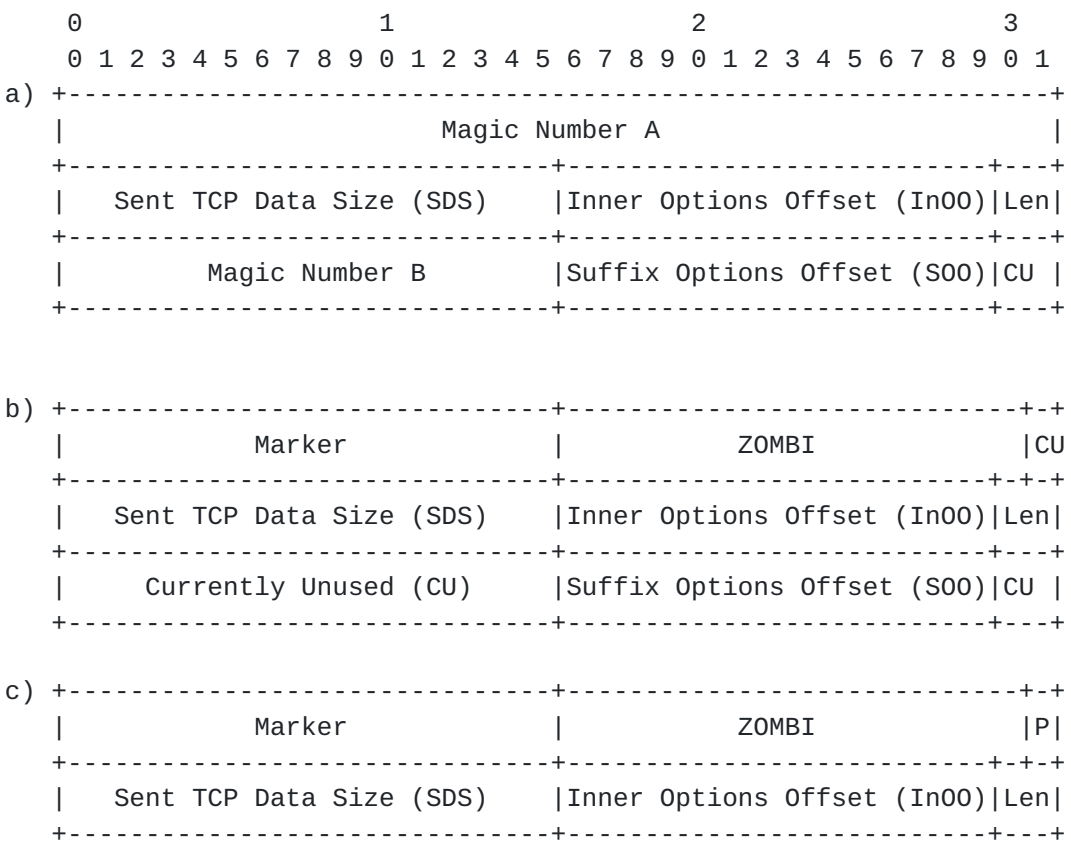


Figure 3: InSpace Option Format a) SYN=1; b) SYN=0, Len=2; c) SYN=0, Len=1

The fields are defined as follows (see Section 3.4 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 excluding the first 4-octet word. In other words, the option is (Len + 1) * 4 octets long. For this experimental specification:

When SYN=1: the sender MUST use Len=2 (12 octets);

When SYN=0: the sender MUST use either Len = 2 (12 octets) or Len=1 (8 octets). If Len = 1, the fields in the last 4-octet word (CU and InOO) are omitted.

Sent Data Size (SDS): In this 16-bit field the sender MUST record the size in octets of the TCP Data when it was sent. This specification defines the TCP Data as all the octets after the TCP Data Offset, including Inner TCP options, the InSpace Option and any padding.

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

Suffix Options Offset (SOO): The 14-bit SOO field defines the offset in 4-octet words from the start of the Inner Options to the start of the Suffix Options. It represents the size of the Prefix Options (see Section 2.3.2).

Prefix (P) flag: The P flag is only defined if Len=1 and SYN=0. In this case the SOO field is not present. Then If P=1, SOO = InOO (i.e. there are only Prefix Options), and if P=0, SOO=0 (i.e. there are only Suffix Options).

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

The following field is only defined within a segment with SYN=1 (i.e. a SYN or SYN/ACK):

Magic Numbers A & B: The sizes of these fields are respectively 32 & 16 bits. The sender MUST fill them with Magic Numbers A & B {ToDo: Values TBA}.

The following fields are only defined within a segment with SYN=0:

Marker: The sender must fill this 16-bit field with zeros (0x00).

ZOMBI: This 15-bit field is used to start encoding or decoding the ZOMBI encoding (respectively see Section 2.3.1.6 or Section 2.3.2.2).

2.3. Inner TCP Option Processing

The objects that Inner Space places within the TCP Data can be divided into two types:

In-Order Flow-Controlled Objects: The receiver processes Suffix Options and the TCP Payload in order, so it might have to buffer them while waiting for a gap in the datastream to be filled by a retransmission. Buffering requires flow control, therefore these will be called In-Order Flow-Controlled objects.

Fire-and-Forget Objects: In contrast, when a segment arrives at the receiver, it never buffers the padding, InSpace Option and any Prefix Options; it immediately processes and removes them. The sender does not need to retransmit these objects if they do not arrive; it creates them on-the-fly to complement each sent

segment. If it has to re-send a segment, it will create new ones relevant to the re-sent segment. Therefore, these will be called fire-and-forget objects.

The rationale for these choices is given in Section 3.2.6. The following two subsections lay out the order in which these options are processed respectively when the sender writes them and when the receiver reads them.

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

The sender MUST add ("3 - ((seqno - isn - 1) % 4")) octets of non-zero padding ("Pad" in Figure 2) to align the start of the InSpace option on a 4-octet word boundary from the start of the datastream, where "seqno" is the TCP sequence number of the segment, "isn" is the initial sequence number and '%' is the modulo operation.

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. The end of the Prefix TCP Options MUST be similarly aligned.

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

2.3.1.3. Sequence Space Consumption

The sender MUST include all the TCP Data in TCP's sequence number and acknowledgement number space, i.e. any padding, the InSpace Option and any Inner Options as well as the TCP Payload.

Whenever the sender includes non-zero TCP Payload in a segment, it MUST also include an InSpace Option, whether or not there are any Inner Options (to enable reconstruction in case of resegmentation).

On the other hand, if the sender includes no TCP Payload in a segment (e.g. ACKs, RSTs), it SHOULD NOT include an InSpace Option unless it is necessary to send an Inner Option. {ToDo: Consider whether there is any reason to preclude Inner Options on a RST, FIN or FIN-ACK.}

A sender MUST consider the sequence space consumed by InSpace options, any padding and any Prefix Options as implicitly acknowledged. Therefore, the sender has no need to hold these items in its retransmit buffer. A sender MUST hold Suffix Options (and TCP Payload, of course) in its retransmit buffer until they are acknowledged.

These rules and those below concerning flow control and pure ACKs have significant implications, which are discussed alongside their rationale in Section 3.2.6.

2.3.1.4. Flow Control Coverage

The sender MUST count Suffix Options and the TCP Payload towards consumption of the receive window advertised by the remote host. Nonetheless, the sender MUST NOT count any padding, the InSpace Option and any Prefix Options towards consumption of the advertised receive window.

There might be a legacy middlebox on the path that discards segments containing out-of-window data but does not understand the way the Inner Space protocol modifies flow control. To traverse such a middlebox, a sending implementation SHOULD use a modified flow control algorithm that avoids the send window dropping below a minimum threshold Snd.Wind.Min (instead of zero). Each sender unilaterally chooses Snd.Wind.Min to allow for Fire-and-Forget Objects it might need in flight on its half-connection. The receiving sides of both half-connections play no part in this allowance. Section Section 3.2.6.2 discusses the rationale for this approach.

A reasonable value for the sender to choose for "Snd.Wind.Min" would be twice the size of the fire-and-forget objects currently in flight.

This would ensure that a middlebox still considers all the fire-and-forget objects are in-window, even if a whole window were lost and retransmitted.

2.3.1.5. Presence or Absence of Flow-Controlled Data

There are three types of acknowledgement segment:

1. An ACK containing no TCP Data is called a Pure ACK;

2. An ACK with no Flow-Controlled Objects (no TCP Payload and no Suffix Options) but some Fire-and-Forget Objects (i.e. an InSpace Option and possibly some padding and Prefix Options) is called an Impure ACK

3. An ACK can be piggy-backed on a segment containing Flow-Controlled In-Order Objects (either TCP Payload or Suffix Options).

It is expected that impure ACKs will rarely be necessary. An example of an Impure ACK is a segment containing no TCP Payload, but still carrying a message authentication code (MAC) in a Prefix Option in order to authenticate and protect the integrity of the TCP header of the ACK.

If an Inner Space TCP implementation currently has no further TCP Payload or Suffix Options to send, and it receives Impure ACKs, it MUST NOT itself respond with further impure ACKs, i.e. it MUST NOT consume further sequence space solely to acknowledge impure ACKs.

Nonetheless, while it has no further TCP Payload or Suffix Options to send, it MAY cumulatively acknowledge the TCP Data in the impure ACKs it has received by emitting a pure ACK, but no more often than once per round trip time (see Section 3.2.6.2 for rationale). If it later starts sending further Payload Data and/or Suffix Options, it will cumulatively acknowledge the sequence space of all the TCP Data in the intervening impure ACKs it has received, as would be expected.

If a sequence of one or more Impure ACKs is dropped, the receiver will not know whether they were impure. The receiver's normal ACK feedback will request a retransmission of the missing sequence space. By definition, the sender does not hold fire-and-forget options in its retransmit buffer. Therefore, the sender MUST reconstruct a new impure ACK of at least the same size as the gap in fire-and-forget options (if SACK has not been negotiated the sender will only know the size of the gap up to any subsequent in-order objects). The sender will include whatever Prefix options are relevant at the time of retransmission (which might be none). If the size of the new

Prefix Options is less than the gap to be filled, the sender MUST make up the shortfall with noop Prefix Options. If the size of the new Prefix Options is greater than the gap to be filled, no harm will be done. This is because the receiver discards fire-and-forget options after processing them, so any overflow will not overwrite flow-controlled in-order data already in the receive buffer.

2.3.1.6. Construction Order for TCP Data

The sender constructs the TCP Data in the following order:

1. It writes any padding, the Inspace Option, Prefix Options, Suffix Options and Payload Data into the TCP Data of the segment.

2. It applies any transformation of the data that might be required, e.g. compression or encryption initiated by a previous control message applied at the TCP layer.

   If SYN=0, and if any such transformation is sensitive to the delivery order of segments, the padding, InSpace Option and Prefix Option MUST remain unaltered (because they need to be processed as soon as they arrive, without waiting to fill gaps in the sequence space).

3. If SYN=0, the sender MUST apply the zero overhead message boundary insertion (ZOMBI) encoding to the segment (see Appendix A).

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, e.g. 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 12 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 Data Size matches the size of the TCP Data.

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.

The receiver MUST NOT count the size of Prefix Options against the receive window. Strictly it ought to subtract the size of Suffix Options from the receive window on arrival, then add the size back again as it removes them. However, when SYN=1, the Suffix Options will never have to be buffered, so these redundant steps can be skipped.

Once only the TCP Payload (if any) remains, the receiver holds it ready to pass to the application. It then emits the appropriate Upgraded Acknowledgement to progress the 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 whether 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 emits the appropriate Ordinary Acknowledgement to progress the 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.

The receiver processes Prefix Options and Outer Options in the order they are received. But it processes Suffix Options in the order they were sent, which is not necessarily the order in which they are received. The receiver achieves this by processing an arriving segment with SYN=0 in the following order. (Steps 3 & 6 are included for completeness even though no current TCP options apply data transformations):

1. It buffers the TCP Data in sequence space order along with any previously buffered data. There might be sequence gaps at this stage.

2. It MUST then ZOMBI decode the buffered data Appendix A. If the stream has not been resegmented, the process is straightforward, but the following steps also check for the more general case where resegmentation might have occurred:

   A. When it finished ZOMBI decoding the immediately preceding TCP Data, the receiver might have run out of data in the middle of a segment and stored the outstanding segment length to decode. If so, the receiver simply continues the unfinished. ZOMBI decoding as long as there is contiguous data to decode.

B. Otherwise, the receiver checks for a 0x0000 marker in the new segment. It starts at the first 4-octet-aligned word in the segment (counting from the ISN). If not present, it scans the TCP Data for the first occurence of such a marker. It classifies any data before the marker as undecoded (conceivably it could find no marker, then the whole arriving segment would remain buffered for later decoding).

C. Starting from the first marker found, the receiver reads the SDS field from the InSpace option and runs the ZOMBI decode algorithm over the extent of the sent data segment. It repeats this for any following sent segments (which might be present due to segment coalescing).

The receiver uses each InSpace Option to calculate the extent of the associated Inner Options (using SOO and InOO).

3. It applies any order-insensitive transformation of the TCP Data that might be required, e.g. counter-mode decryption initiated by a previous control message applied at the TCP layer:

4. It MUST then remove the InSpace Option and it MUST process and remove TCP options in the following order:

A. It processes and removes any Prefix TCP Options. (During the decoding process the receiver might find Prefix Options on multiple sent segments within a single newly arrived segment, due to prior resegmentation.) Note: it does not subtract the size of Fire-and-Forget Objects from the receive window.

B. It processes and removes any Outer TCP Options of the newly arrived segment (note that if an arriving segment contains multiple sent segments, the receiver processes all the Prefix Options within it before processing any Outer Options).

C. It buffers Suffix Options and TCP Payload, subtracting from the receive window ("Rcv.Wind") accordingly.

5. It emits an ACK if appropriate (typically using regular TCP ACK behaviour, but see Section 2.3.1.5 concerning Impure ACKs).

6. Once gaps (if any) in the datastream have been filled, the receiver applies any order-sensitive transformation of the TCP data that might be required, e.g. decompression or decryption initiated by a previous control message applied at the TCP layer:

A. The TCP receiver MUST apply an order-sensitive transformation progressively, to one sent segment at a time in sequence

order from the start of one Payload up to the end of the next set of Suffix Options (which might change the way it transforms the next segment, e.g. a rekey option).

B. Having established the extent of the next sent segment, The receiver returns to step 6A.

7. It processes and removes any Suffix Options strictly in datastream order, as illustrated in Figure 4a) in Section 3.2.6. It adds to "Rcv.Wind" accordingly.

Once only the TCP Payload remains, the TCP receiver passes it to the application as normal.

2.3.3. Forwarding Inner TCP Options

Middleboxes exist that process some aspects of the TCP Header. 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. Section 3.2.3 explains how the endpoints will be able to force middleboxes to comply with this rule once they can authenticate of even encrypt TCP options within the TCP Data, whereas if they tried to enforce this rule today they would only damage their own transmissions. 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 Data Size greater than 2^16 - 21 (=65,515 = 0xFFEB) 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 Data when it was sent. This is because the TCP Data in a regular IPv6 packet cannot be greater than (2^16 -1 - 20) octets (given the minimum TCP header is 20 octets). If the size of the TCP Data is greater than 0xFFEB octets, the sender MUST use a Jumbo InSpace Option (Appendix C.2).

A Sent Data 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.

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 7).

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];

- and trying SCTP and TCP in parallel [I-D.wing-tsvwg-happy-eyeballs-sctp];

- and trying ECN and non-ECN in parallel;

- 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. Inner Option Space

3.2.1. Header Extension by Encapsulation

It has been proposed [Briscoe14] that extension of a header (as opposed to options) at layer X ought not to be located within the header at layer X, but instead within the layer encapsulated by that

header (layer X+1), for a selection of principled and pragmatic reasons:

1. Implementations of layer X that have not implemented or are not interested in an extension to layer X need not be bothered with walking over a load of extensions they do not know or care about.

2. An extension always requires a new implementation, which can be coded to know where to look for the extensions it implements; extensions never need to be located where unmodified code can find them.

3. Layer-X middleboxes that do not correctly forward layer-X extensions are common, but they do tend to forward their layer-X+1 payload correctly. Therefore extending layer-X within an encapsulation is more likely to traverse badly designed middleboxes.

4. Extension by encapsulation is not a manifesto for extending layer X at layer X+1, X+2,... and ever-deeper. Usually a base protocol design is sound, and an an extension is not permanently necessary to make it fit for purpose; the extension merely adds something needed in circumstances not originally conceived. Therefore it is rare that an extension becomes so ubiquitous that extensions to the extension become necessary.

5. Extending layer X within a layer-X+1 encapsulation should not be confused with an attempt to evade security middleboxes. If an attack on layer X is encapsulated in layer X+1, security middleboxes will be reprogrammed to block it. Whereas, if a useful extension to layer X were encapsulated in layer X+1, security middleboxes would not be reprogrammed to block it.

6. If the endpoints of layer X don't want layer-X middleboxes to intervene in their layer-X extension, they can encapsulate it within layer X+1. In contrast, if they want an extension for co-operation with layer-X middleboxes, they can place it in the layer-X header. Then everything at layer X+1 can be authenticated and/or encrypted to structure and enforce the distinction between the types of extension, without having to selectively authenticate and/or encrypt parts of the layer X header.

3.2.2. 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:

- about a quarter appeared to actively remove options that they did not recognise (perhaps assuming they might be indicative of an attack?);

- the rest were some type of higher layer proxy that split the TCP connection, unwittingly failing to pass unknown options between the two connections.

The magic number approach ensures that all the TCP Headers and options up to the Data Offset are completely indistinguishable from an Ordinary Segment. Therefore, it will be highly likely (but not certain--see Appendix C.1.4) that the extra Inner Options will always be forwarded, while the conventional approach would fall far short of ths ideal.

The magic number approach also ensures that the Inner Options and the option that points to them are both tucked away beyond the Data Offset (see Section 2.2.1). This makes it highly likely that the two will share the same fate--it would be extremely unusual for a middlebox to treat different parts of the TCP Data selectively.

Typically, if a TCP option were stripped, the concern would only be lack of function, not safety. But with option space extension, the concern is serious application corruption. If control options are placed beyond the Data Offset, and the option that says they are there gets stripped, it risks control options being passed to the application as (corrupt) data. Although option stripping can be detected during the handshake, this consumes round trips and it is does not guarantee that option stripping will not start part-way through a connection (e.g. due to a path change). In contrast the magic number approach is inherently safe.

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 Data 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 perfect 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.

3.2.3. 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.

3.2.4. 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 C.3. 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.

3.2.5. Framing Segments

A middlebox that splits a TCP connection can coalesce and/or divide the original segments. Segmentation offload hardware is another common cause of resegmentation. Inclusion of the marker in the InSpace Option allows the receiver to reconstruct the original segment boundaries. The ZOMBI encoding Appendix A removes any occurrences of the marker other than those at the start of each segment.

Superficially, the receiver does not need the sent data size (SDS) field to find the end of each sent segment; it could scan for the marker at the start of the next segment instead. However, in the common case when a stream has _not_ been resegmented, the receiver will find the marker at the start of the segment, but the next marker will not have been received yet. The SDS field allows the receiver to know immediately whether a whole segment has been received as sent. For the same reason, Minion [I-D.iyengar-minion-protocol] uses a (different) marker to tag the end of each message. In contrast, the Inner Space approach uses 2B to declare the original segment size, which saves having to scan the stream for an end marker.

Equally, one could argue that markers are unnecessary, because the sequence of sent data size fields from the start of the stream seem sufficient to find all the segment boundaries. Using markers ensures that the receiver can pick out segment boundaries immediately on arrival, which is important for deadlock avoidance (see Section 3.2.6).

The Sent Data 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 C.3 if it is needed.

3.2.6. Control Options Within Data Sequence Space

Section 2.3 introduced the two types of objects that Inner Space places within the TCP Data:

In-Order Flow-Controlled Objects: Suffix Options and the TCP Payload;

Fire-and-Forget Objects: Padding, the InSpace Option and any Prefix Options.

The following two sections address each in turn: i) explaining why it is useful to introduce in-order flow-controlled TCP options and ii) explaining why it is feasible to encapsulate fire-and-forget options within the TCP datastream, despite its reliable ordered semantics.

3.2.6.1. In-Order Flow-Controlled Options

Including Suffix 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 except a no-op Suffix option. 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.

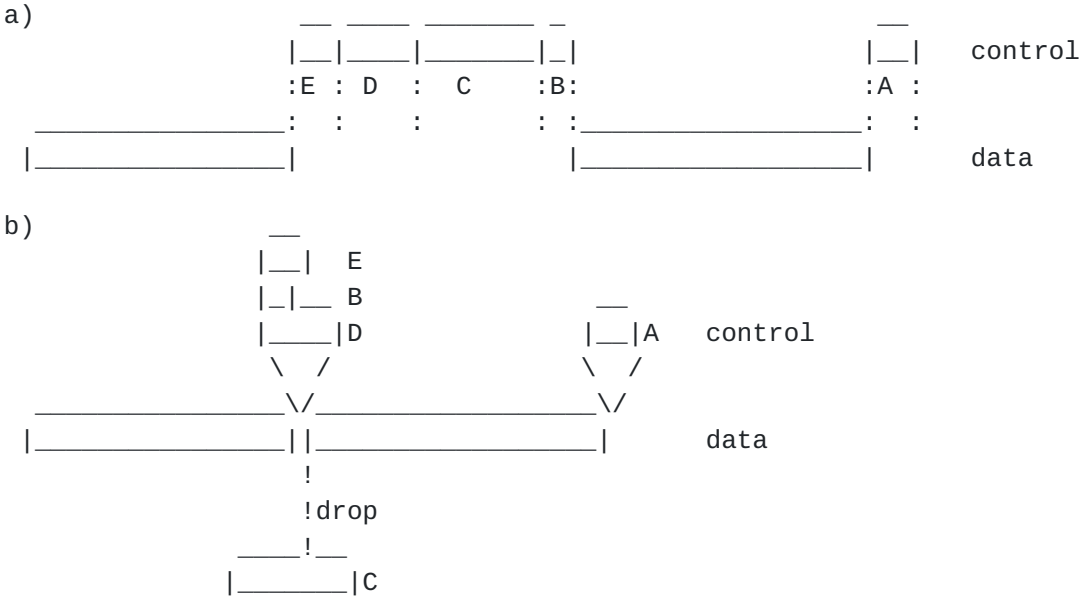


Figure 4: Control options a) inside vs. b) outside TCP sequence space`

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-tcpcrypt]).

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.

3.2.6.2. Fire-and-Forget Options

The Inner Space protocol allows Fire-and-Forget Options to be tunnelled within the TCP Data so that they can traverse middleboxes that would otherwise strip them or somehow normalise their contents. Two question then arise: i) should Fire-and-Forget Objects (padding, the InSpace Option and Prefix Options) consume sequence space and ii) should they be covered by flow control? The answers to these questions will also be re-usable to multiplex streams within one TCP connection:

Sequence Space: Ideally, fire-and-forget objects would not consume sequence space, because they do not need to be retransmitted. However, many middleboxes expect the TCP sequence number to increment consistently with the amount of TCP Data. For instance, a split connection would be likely to 'normalise' sequence numbers, being unaware that certain items in the datastream might be exempt from sequence space consumption.

Therefore, although it is not elegant, the sender has to consume sequence space for fire-and-forget objects, but it implicitly considers these octets to be immediately acknowledged. And the receiver does not have to immediately acknowledge sequence space consumed solely by fire-and-forget objects; it can defer until it acknowledges reliably delivered flow-controlled objects--when it does no harm to cumulatively acknowledge intervening fire-and-forget objects as well. This is the underlying principle behind the normative rules given on sequence space consumption and ACK withholding in Section 2.3.1.3 and Section 2.3.1.5.

Flow Control: The sender does not need to count Fire-and-Forget Objects against the receive window ("Rcv.Wind"), just as it does not count Outer TCP Options against "Rcv.Wind".This should work because It is impossible for middleboxes to 'normalise' the receive window and flow control, because they cannot know when the application is releasing data from the receive buffer. Also the receiver always processes Fire-and-Forget Objects immediately without buffering them; it could be considered that the receiver effectively subtracts their size from "Rcv.Wind" then immediately restores "Rcv.Wind" to its former value.

In fact, as shall now be explained, it has to be _mandatory_ for the sender not to count fire-and-forget objects against "Rcv.Wind". It is important for deadlock avoidance that certain TCP options never consume "Rcv.Wind". Some TCP options acknowledge data, e.g. SACK or the Data ACK within the Data Sequence Signal (DSS) sub-option of MPTCP. Other TCP options need to be applied to all ACKs, e.g. the MAC of tcpcrypt. If an acknowledgement were to need sufficient advertised receive window before it could be sent, there would always be a risk of deadlock if the receiver ever needed the acknowledgement before it could release more receiver buffer [Raiciu12].

The rule above concerning sequence space is a compromise needed to traverse middleboxes. So, perhaps predictably, this begets further compromises. The rule concerning flow-control is principled. So perhaps predictably, it has to be compromised to traverse certain middleboxes. The rationale for these compromises is explained below, referring to the normative rules in the protocol specification where appropriate:

Sequence Space: If the sender does not retransmit unacknowledged data after a RTO, some middleboxes will mimic TCP's retransmission timeout (RTO) and resend the fire-and-forget data themselves, which could lead to an ACK storm. Therefore, Section 2.3.1.5 allows a receiver to emit a pure ACK every round trip, just to keep such middleboxes quiet. In general, allowing TCP to ACK an ACK can lead to an ACK storm. However, in this case, all that is allowed is a Pure ACK in response to an Impure ACK, which immediately terminates any potential for a vicious circle. This solution even works in the case where both TCP hosts ignore ACKs unless they are authenticated (which the pure ACK will not be). No harm will arise if the remote host ignores the pure ACK, because it is only for the benefit of a middlebox anyway.

If a sequence of one or more Impure ACKs is lost the receiver cannot suppress retransmission, because it can only decide whether it needed in-sequence data once it arrives. Therefore, loss of

fire-and-forget data causes a retransmission that may prove to be unnecessary. By the rules in Section 2.3.1.5, an ACK would only include fire-and-forget data in the first place if it was actually necessary. Therefore, normally retransmission of Impure ACKs will be required and useful. However, sometimes, the Prefix Option(s) within the Impure ACK(s) might have become unnecessary. This inefficiency could just be ignored, or partial reliability could be added to TCP to address it. The Inner Space protocol does not prevent partial reliability being added, but it does not require it either.

Flow Control: Some middleboxes attempt to mitigate scanning or DoS attacks by reading the window field in the main TCP header (and the Window Scale outer TCP option if present) and discarding segments that they calculate contain data that is out-of-window.

Section Section 2.3.1.4 requires the two endpoints to tacitly agree that the fire-and-forget portion of the TCP Data is exempt from flow control. A legacy middlebox will not know this, so it might think data is out-of-window when the endpoints have agreed it is in-window. Section Section 2.3.1.4 provides a solution to this problem, which is only necessary if a TCP implementation is deployed where there is a risk of encountering such middleboxes. The solution involves the TCP sender denying itself the use of the bottom of the buffer advertised by the receiver. Normally the sender stops sending when it calculates the remaining receive window is zero. Instead, the modified sender sets itself a threshold (Snd.Wind.Min) to allow for the Fire-and-Forget Objects it might need in flight, and it stops sending before the receive window drops below this threshold.

Snd.Wind.Min bytes at the 'left-hand' end of the receive buffer are wasted by this solution (to be fair, the middlebox behaviour is really to blame). An alternative was considered where the sender and receiver use a new Inner TCP Option to agree a window offset between themselves, so that middleboxes are not party to their agreement. Although, this would not waste any of the left-hand end of the receive buffer, it would reduce the maximum advertised buffer at the right-hand end by the same amount. Therefore the sender-only solution was chosen, given it is much simpler, and the sender can continuously adapt how much allowance it sets aside throughout the connection, rather than having to commit to a necessarily conservative estimate at the start.

3.3. Deployment Approach

3.3.1. Substrate Protocol: TCP vs. UDP

Inner Space uses TCP as a substrate protocol, i.e. on the wire, the headers look like an RFC793-compliant TCP, and there is only a difference if one looks inside the TCP Data. Other transport extensibility approaches have used UDP as a substrate protocol, for instance, to carry SCTP through middleboxes.

In design and implementation terms, it is much easier to turn UDP into a reliable protocol, than it is to selectively turn TCP into an unreliable protocol. However, UDP is already blocked on about 15% of Internet paths {ToDo: ref}, whereas vanilla TCP is still universally permitted. Therefore, because the goal is middlebox traversal, not just ease of implementation, Inner Space uses TCP as a substrate.

It may well turn out that Inner Space cannot reach some places that UDP can. It is expected that applications (or even the TCP stack) might sometimes have to resort to tryinging UDP as a substrate in such cases.

3.3.2. Kernel-Space vs. User-Space

At an earlier stage in the specification of the Inner Space protocol [I-D.briscoe-tcpm-inner-space] before unordered delivery of Inner Options was introduced, Inner Options could all be processed in either user-space or kernel-space. The only exception was the interactions controlling the handshake on the first segment in each direction. However, with the addition of unordered delivery of Prefix Options, the protocol has to be implemented in the kernel, because the protocol modifies the behaviour of TCP, not just its payload.

3.4. 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 option is always known by the marker in the InSpace option as well as by the offset from the previous one, using the Sent Data Size field.

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 8 octets from the left-hand edge of the marker that starts the InSpace Option. From the Length, the receiver can always determine the layout of the rest of the option. The length is in 4-octet words because the InSpace option is always a multiple of 4 octets long, so that any subsequent Inner TCP Options comply with TCP's option alignment requirements.

Sent Data 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 C.2, which offers a 32-bit Sent Data 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.

Inner Options Offset: This field is in units of 4-octet words, so its width is 14-bits. Then, if necessary Suffix Options can be as large as a maximum sized segment (given 4 * 2^14 = 2^16 octets).

Suffix Options Offset: The InOO field is the same 14-bit width as the SOO 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.

When SYN=1 the layout of the InSpace Option includes:

Magic Numbers: 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, Magic Number B provides 16 more bits of magic number. Otherwise, these 16-bits would only have to be used for padding to align with the next 4-octet word boundary anyway.

When SYN=0, the following further considerations determined the layout of the InSpace Option:

ZOMBI: The ZOMBI field holds an offset that has to be sufficiently wide to span the extent of a maximum-sized segment of 2^16 bits.

Given the offset is measured in 2-octet units, this means the ZOMBI field has to be at least 15 bits wide (see Appendix C.2 for the size of the ZOMBI field for a jumbogram).

Marker: Given occurrences of the marker are replaced by offsets of the size of the ZOMBI field, the marker has to be at least as wide as the ZOMBI field. However, a 16-bit marker is used, because it is more efficient than having to replace 15-bit markers.

Currently Unused (CU): There are three CU fields in the InSpace option when SYN=0 that fill odd corners of space. Unfortunately, this is necessary to ensure 4-octet alignment of the first Inner Options.

Prefix (P) flag: When there are solely Prefix Options, or solely Suffix Options, a short-form InSpace Option can be used (Len = 1) by omitting the last 4-octet word. Then the P flag determines whether there are solely Prefix Options or solely Suffix Options in the Inner Options field. Whenever both Prefix and a Suffix Option are needed on the same segment, even though only 14 more bits of framing information are needed, the InSpace option has to grow in steps of 32 bits to maintain 4-octet alignment. Therefore 18 bits have to be assigned as Currently Unused (CU).

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 two, if dual handshakes are used.

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 (h+60B+12B) where h is the IP header size (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 12B). That is H will be 92B for IPv4 or 112B for IPv6;

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*8/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 assumes an InSpace option adds 8B per segment (i.e. both Prefix and Suffix Options together on every segment will be rare). For example, keeping P as it was in the above example and taking Q=10% and F=750B, the traffic overhead is 0.09%. It is as difficult to predict Q as it is to predict P.

Processing: {ToDo: Implementation tests}

5. Interaction with Pre-Existing TCP Implementations

5.1. Compatibility with Pre-Existing TCP Variants

It is believed that all TCP options that were designed as Outer Options can be relocated without alteration as Prefix Options, because the unreliable unordered semantics are the same as TCP Outer Options. However, some yet-to-be-defined TCP options might be better suited to the reliable ordered semantics of Suffix Options. Specifically, existing or proposed TCP options fall into the following categories:

Segment-Related: Concerned with the delivery of individual segments as they arrive at the receiver. Therefore these options MUST NOT be located as Suffix Options:

* Timestamp [RFC7323] on SYN=0 segments;

* SACK [RFC2018];

* The Data ACK part of the DSS option of Multipath TCP [RFC6824];

* TCP-AO [RFC5925] if covering TCP Options;

Stream-Related: Controlling delivery of an ordered stream. Therefore these options SHOULD be located as Suffix Options:

* The tcpcrypt CRYPT sub-options [I-D.bittau-tcpinc-tcpcrypt].

Connection-Related: Controlling the parameters of a connection. These options can be located either as Suffix, Prefix or Outer Options:

* No-op and end of option list [RFC0793];

* Maximum Segment Size (MSS) [RFC0793];

* SACK-ok [RFC2018];

* The timestamp when used on SYN=1 segments to indicate support for timestamps [RFC7323];

* 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];

{ToDo: The above list is not authoritative. Some TCP options include suboptions, some of which are discussed below, but others remain to be fully assessed.}

The specification of any future TCP option MUST state whether it is designed as a Suffix Option (reliable ordered) or as a Prefix / Outer Option (unreliable unordered) or "Don't Care". A TCP option MUST by default only be used as an Outer or Prefix Option, unless it is explicitly specified that it can (or must) be used as a Suffix Option.

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 Suffix or Prefix 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 Suffix Options (the Checksum would not be necessary).

The MPTCP Data ACK has to remain as a Prefix or 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 a Suffix 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 authentication option can be configured either to cover TCP Options or not (when it was defined only Outer Options existed). If it covers any TCP Options it has to be located as an Outer or Prefix Option to prevent the possibility of flow-control deadlock (because it would consume receive window on pure ACKs if it were located as a Suffix Option).

All sub-options of the tcpcrypt CRYPT option could be located as Suffix 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.}

5.2. 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 extending TCP within the TCP Data;
  - Section 3.2.2 justifies the magic number approach;
  - Section 3.2.3 explains why the protocol will remain robust as middlboxes evolve;
  - Section 3.2.6 justifies including Inner Options in sequence;

* Section 3.2.5) explains how the protocol will remain robust to resegmentation.

5.3. 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 placement of options within the TCP Data 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 can be used, but the dual handshake MUST be disabled, because the dual handshake has to try two ports. Therefore, if the app binds to a specific port, the upgraded SYN MUST be tried first on its own, then if that reveals that the server is not upgraded, the stack will abort that connection with a RST and use the same port to send an ordinary SYN. Use of a specific port might be necessary, for example in the FTP protocol, 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.

Some applications interrogate the TCP stack to determine the path max transmission unit (PMTU), e.g. in order to optimize application message boundaries within the datastream. From the viewpoint of such applications, TCP options subtract the same amount from the PMTU whether they are Outer or Inner Options. However, the 8 (or 12) octet InSpace header and the alignment padding represent extra overhead. Therefore, for such applications, the TCP stack as seem through the socket API will seem similar to a tunnel that reduces the useful size of the PMTU. This could lead to fragmentation until such applications are updated. Nonetheless, most such applications already include code to adapt to PMTU reduction by tunnels.

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 how much space is available, which depends on whether Inner Space is enabled.

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 D). 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.

6. 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);
- 0xMMMM (16 bits);

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

7. 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 allow attackers to exhaust the TCP server's memory more easily. 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.

Some security devices block data in an initial SYN segment, classifying it as the signature of an attack. Attackers might indeed use data-in-SYN to strengthen the force of a SYN flood attack, but it has also always been valid for clients to use data-in-SYN for low latency service as well (today data-in-SYN is used by TCP Fast Open, but data-in-SYN has been permitted for similar reasons right back to the days of RFC 793). On its own, data-in-SYN MUST NOT be considered a sufficient signature of an attack. It can only be considered an attack signature if seen in combination with other symptoms of a SYN flood attack. The logic that led to data-in-SYN alone being considered an attack was probably well-intentioned, but it actually turns a security device into an attack on innocent low latency services.

The optional extension for DPI traversal specified in Appendix C.3 might create a new attack vector. The attack was originally proposed (by David Mazieres) when an earlier draft required the optional extension to be applied at the start of both half-connections. As long as the DPI traversal extension no longer applies in the server-client direction the attack seems less feasible. Nonetheless, the attack in the server-client direction is described here anyway (in

case it prompts someone to think of a similar feasible attack in the client-server direction):

Attack that used to be feasible in the server-client direction: An attacker could have crafted content (e.g. a binary file such as a graphics object) such that it included the appropriate bits in the correct positions to match the Inner Space magic numbers and the expected format of some TCP options. It could have then uploaded this content to a legacy server for download by other clients (e.g. a public image archive). Then, if an upgraded Inner Space TCP client had accessed this legacy server, it would have seemed as if the server was upgraded. So the attacker could have theoretically conscripted the server into sending TCP options of its choice. Although the attacker would have been limited to TCP options relevant to those previously proposed by the client, some harm might have been possible. The attacker might also have been able to contrive the remainder of the content (after removing the apparent TCP options) to be some form of script or executable.

If the DPI traversal solution is to be used, and a feasible attack is developed in the client-server direction, a couple of directions to prevent such an attack could be explored:

- the magic number would somehow have to be complemented by another signal, perhaps out of band;

- the magic number would need to somehow include a cryptographic hash of material sent by the client, so that an attacker could not predict it.

8. 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. Jana Iyengar also suggested the sender-only flow-control offset. 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 comments: Joe Touch, Yuchung Cheng, John Leslie, Mirja Kuehlewind, Andrew Yourtchenko, Costin Raiciu, Marcelo Bagnulo Braun, Julian Chesterfield, Jaime Garcia, Ted Hardie and David Mazieres, Tim Shepard, Mark Handley.

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.

9. References

9.1. 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.

9.2. Informative Reference

[Briscoe14]
Briscoe, B., "Tunnelling through Inner Space", IAB Workshop on Stack Evolution in a Middlebox Internet , January 2015.

[Cheshire97]
Cheshire, S. and M. Baker, "Consistent Overhead Byte Stuffing", Proc. ACM SIGCOMM'97, Computer Communication Review 27(4):209--220, October 1997.

[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-tcpcrypt]
Bittau, A., Boneh, D., Hamburg, M., Handley, M., Mazieres, D., and Q. Slack, "Cryptographic protection of TCP Streams (tcpcrypt)", draft-bittau-tcpinc-tcpcrypt-00 (work in progress), October 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.briscoe-tcpm-inner-space]
Briscoe, B., "Inner Space for TCP Options", draft-briscoe-tcpm-inner-space-01 (work in progress), October 2014.

[I-D.ietf-httpbis-http2]
Belshe, M., Peon, R., and M. Thomson, "Hypertext Transfer Protocol version 2", draft-ietf-httpbis-http2-17 (work in progress), February 2015.

[I-D.iyengar-minion-protocol]
Jana, J., Cheshire, S., and J. Graessley, "Minion - Wire Protocol", draft-iyengar-minion-protocol-02 (work in progress), October 2013.

[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.

[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.

[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. Zero Overhead Message Boundary Insertion (ZOMBI)

This appendix is normative and mandatory to implement for the Inner Space protocol. This encoding is relegated to an appendix merely because it is applicable more generally than for just Inner Space. Therefore, in a future revision, this appendix might be removed and replaced by a reference to a stand-alone document.

The Inner Space protocol requires the sender to add a marker in every segment at the first 4-octet aligned word from the start of the datastream. Then, even if the stream is subsequently resegmented, the receiver can recover segments and their associated TCP options as they were sent. The sender uses the value 0x0000 as the 2-octet marker at the start of the InSpace option header. It uses the ZOMBI encoding to remove all other occurrences of 0x0000, treating the segment as a sequence of 2-octet shorts. Then, a marker will unambiguously locate the InSpace option at the start of each segment. From this InSpace option, the receiver can find the length of the segment. Then it can decode the ZOMBI encoding to return the segment to its original form.

The sender applies the ZOMBI encoding as follows:

1. It places 0x0000 in the Marker and the ZOMBI fields of the InSpace option, and fills all the other fields of the InSpace option with the relevant sizes and offsets.

2. Treating the stream as a sequence of 2-octet shorts,starting from the ZOMBI field, it replaces each occurrence of 0x0000 with the offset (in shorts) to the next occurrence of 0x0000, or to just beyond the end of the segment when there are no more occurrences of 0x0000.

Because an offset can never be zero, this process naturally removes all occurrences of 0x0000 from the segment.

The receiver reverses the above encoding, assuming the worst case of a resegmented stream unless it finds otherwise:

1. If it is buffering undecoded bytes either side of the newly arrived segment in the sequence space, it coalesces them.

2. Scanning two octets at a time aligned on even numbers of octets from the ISN, it locates the next occurrence of an InSpace option by locating the next occurrence of 0x0000 in a segment.

3. Starting at the ZOMBI field, it points a variable (e.g. "ptr") to a position in the stream, reads the short at that location, writes 0x0000 into the stream to replace it, then increments "ptr" by the value just read. It continually repeats the same read, replace and increment operations at each new location pointed to by "ptr".

4. The receiver knows the size of the sent segment from the SDS field, so that it knows when to stop decoding. If the end of the received segment is reached before this, it implies the stream has been resegmented and the next segment has not been buffered yet. In this case, the receiver stores how much decoding is left.

5. If there are more undecoded octets buffered, the process repeats from step 1.

Below an implementation of the ZOMBI encode and decode algorithms is given in C. The decode algorithm would be preceded by marker-scanning code to find the location of the ZOMBI and SDS fields within the InSpace option. The SDS field will always be non-zero, therefore it will never be changed by the encoding, so the receiver can read it before starting to decode. In case length is odd, a non-zero pseudo-padding octet is considered to be appended to the segment while encoding or decoding (but it is not actually transmitted).

```
/* {ToDo: Test}
 * ZombiEncode encodes "length" bytes of data
 * starting directly after the marker pointed to by "ptr", where:
 *   length = sds - pad.
 */

void ZombiEncode(unsigned short *ptr, unsigned short length)
{
  const unsigned short *end = ptr + ++length>>1;  % /2 rounded up
  unsigned short *code_ptr = ++ptr;               % point to ZOMBI
  unsigned short code = 0x0001;

  while (++ptr < end) {                   % initialise after ZOMBI
    if (*ptr == 0) {
      *code_ptr = code;
      code_ptr = ptr;
      code = 0x0001;
    } else
      code++;
  }
}

/* {ToDo: Test}
 * ZombiDecode decodes "length" bytes of data
 * starting after the marker pointed to by "ptr", where
 *   length = sds - pad.
 * Returns number of shorts still to decode.
 */

short ZombiDecode(unsigned short *ptr, unsigned short length)
{
  const unsigned short *end = ptr++ + ++length>>1;  % /2 rounded up
  while (ptr < end) {                         % initialise to ZOMBI
    code = *ptr;
    *ptr = 0;
    ptr += code;
  }
  return (ptr - end);
}
```

The ZOMBI encoding always uses a marker that is larger than the maximum possible segment size. Therefore, for a jumbo segment Appendix C.2, the sender uses 0x00000000 (4 octets of zeros) as the marker; it pads the segment to a multiple of 4 octets; and it scans the stream in 4-octet words, replacing any occurrences of the marker with the offset in 4-octet words to the next marker.

The ZOMBI encoding is similar to consistent overhead byte stuffing (COBS [Cheshire97]). The main difference is that COBS markers are only one octet. Therefore, in COBS, whenever the distance between zero-bytes is greater than 0xFE, it has to insert an extra byte into the stream with the special value of 0xFF. When decoding, 0xFF is removed rather than replaced by 0x00. Therefore, as well as 2 extra delimiting octets, COBS introduces a variable number of extra octets, but no more than 1 in 254 (a more accurate name would have been _capped_ overhead byte stuffing, because the overhead is variable, not consistent).

In contrast, ZOMBI introduces a predictable overhead of 4 delimiting octets per segment (or 5 for odd length segments), with no unpredictable variation. Therefore, space for the known overhead can be set aside in the InSpace option, and the ZOMBI encode and decode operation can be zero-copy, which is not possible with COBS. A more accurate name for ZOMBI would have been _constant_ overhead message boundary insertion. Nonetheless, the encoding to replace markers once the message boundaries have been inserted actually is zero overhead, so the cool acronym is not totally contrived.

Appendix B. Generic Connection Mode Switching

This appendix is normative and mandatory to implement for the Inner Space protocol. This encoding is relegated to an appendix merely because, in a future revision, this appendix might be removed and replaced by a reference to a stand-alone document. 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.

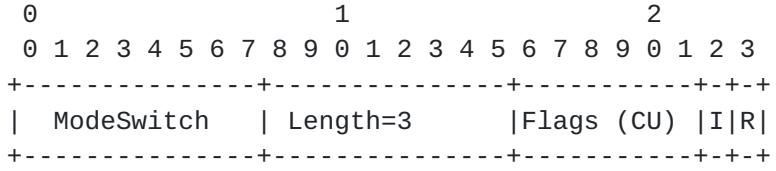


Figure 5: The ModeSwitch TCP Option

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 Data Size (see Section 2.4).

Appendix C. 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.}

C.1. 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 D 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.

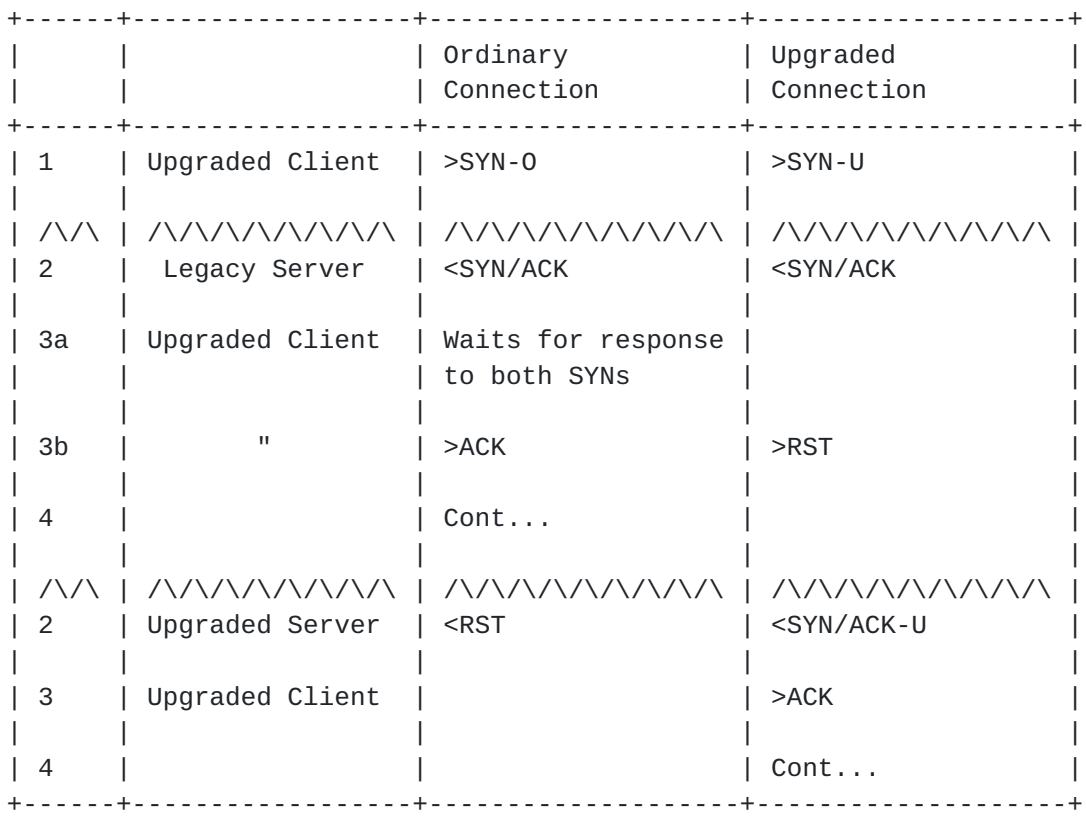


Table 2: Explicit Variant of Dual 3-Way Handshake in Two Server Scenarios

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.

C.1.1. 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.

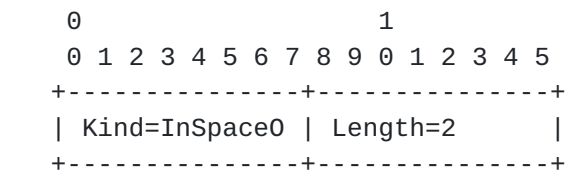


Figure 6: An InSpaceO TCP Option Flag

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].

C.1.2. 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.

C.1.3. 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 C.1.4. 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.

C.1.4. 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 C.1.3) 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.

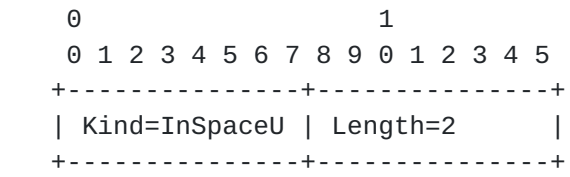


Figure 7: An InSpaceU Flag TCP option

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.

C.2. Jumbo InSpace TCP Option (only if SYN=0)

This appendix is normative. It defines the format of the InSpace Option necessary to support jumbograms. 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, 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 Sent Data Size (SDS), the Inner Options Offset (InOO) and the Suffix Options Offset (SOO) use more bits, respectively 32, 30 and 30. The Length (Len) field can be either 2, 3 or 4, where binary 00 represents 4.

If Len=3: the last 4-octet word is omitted and the value of SOO is determined by the P flag as already described in Section 2.2.2.

If Len=2: it is assumed InOO = SOO = 0.

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.

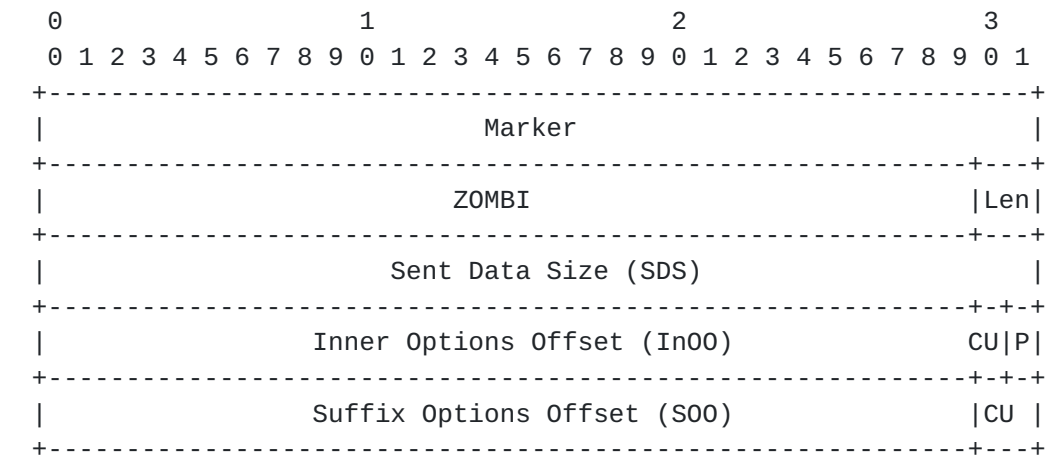


Figure 8: InSpace Option for a Jumbo Datagram

C.3. Optional 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.

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 extension to the

specification has been defined in case experiments prove that it significantly improves traversal of such deep packet inspection (DPI) boxes.

This extension places the expected app-layer headers at the start of the TCP Data in the SYN and in the first data segment in the client-to-server direction:

SYN=1: The sender uses the structure in Figure 9a) on the SYN. The sender right-aligns the 12-octet InSpace Option at the end of the segment. Then it right-aligns the Inner Options against the InSpace Option, all after the end of the TCP Payload and any padding necessary to align the options on a 4-octet word boundary.

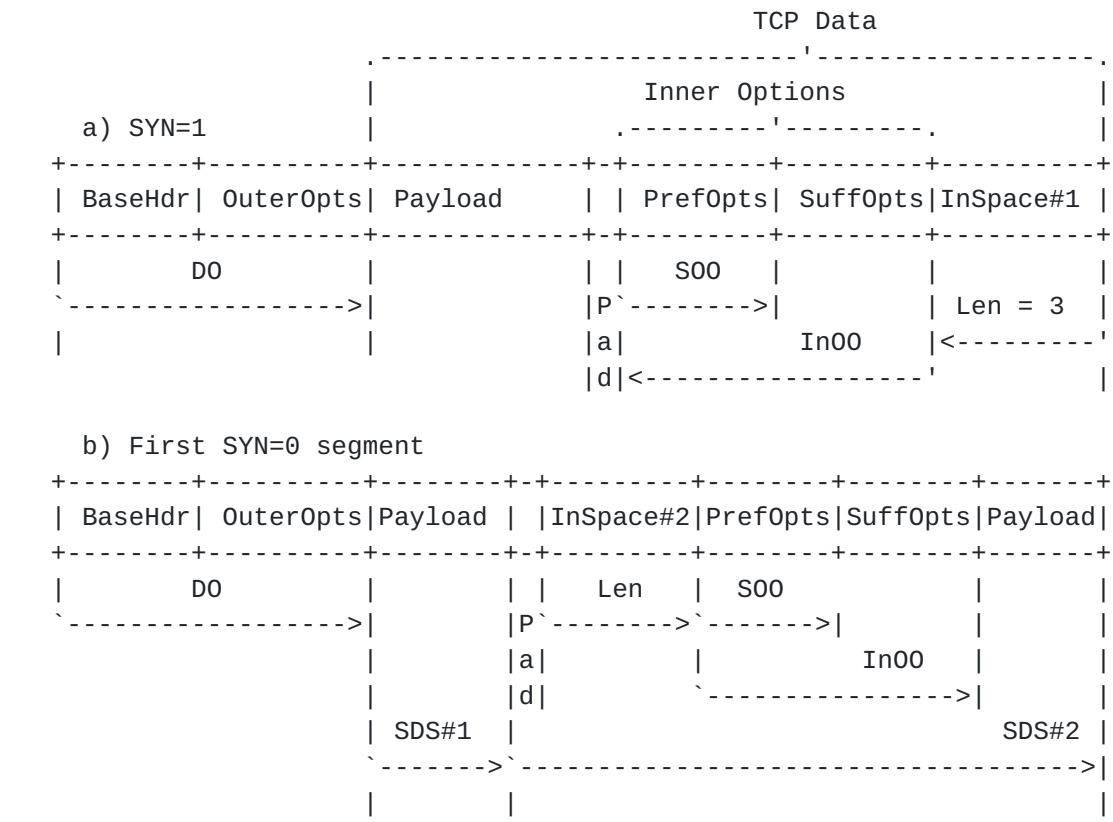
Magic Number A starts 4*3=12 octets from the end of the segment {ToDo: Magic Number A could be placed at the end of the segment instead.}. A receiver implementation of this optional extension MUST check whether Magic Number A is present within the InSpace option if it does not first find it at the start of the segment.

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. 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 SDS#1 octets from the start of the segment (plus possible padding), where SDS#1 is the value of Sent Data 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, 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 Data Size (SDS#2) in the second InSpace Option (InSpace #2) defines the length of the remaining TCP Data before the end of the first data segment, as shown.



All offsets are specified in 4-octet (32-bit) words, except SDS and Pad, which are 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 segment processing more involved. Experimental implementation of this approach will determine whether the extra complexity improves DPI box traversal sufficiently to make it worthwhile.

If it does work, it is believed that this extension will only be necessary on the initial SYN and the first data segment sent in the direction from TCP client to server. Therefore, the SYN/ACK and data segments sent by the TCP server will continue to use the regular Inner Space segment structure illustrated in Figure 2.

If a TCP client that implements this extension opens a connection with a server that does not, the client will fall back to ordinary TCP even though the server would have supported the Inner Space protocol without the DPI traversal extension. This is because the server does not look for the magic number at the end of the SYN, so it behaves like a legacy TCP server responding with an ordinary SYN/

ACK, which in turn makes the client fall back to ordinary TCP. Such limited fall-back is considered sufficient to support experiments to see whether the DPI traversal extension is useful. If it is useful, a future standards track specification could make support for this DPI traversal extension mandatory for an Inner Space TCP server, but still optional for an Inner Space TCP client.

Appendix D. Comparison of Alternatives

D.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 C.1) 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 (some firewalls and middleboxes discard packets with unrecognised TCP options [Honda11]). 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 E. 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.

Data in SYN middlebox traversal: Certain middleboxes do not forward data in a SYN. The scheme can detect this (by the lack of acknowledgement of the data on the SYN/ACK). However, it would be ideal to be able to work round this problem in all circumstances, not just those in Appendix C.1.4.

Options that alter the main TCP header semantics: Need to include text to ensure Inner options are used with care where middleboxes are known to use a main header field, particularly if the middlebox also understands how a TCP option alters its semantics. Examples:

WScale: Easiest to only locate this as an Outer Option - too many TCP normalisers that check whether a segment is in window use WS to interpret the Window field.

SACK: A similar but different example is where a middlebox shifts the ISN, and also shifts all seqno values including in TCP options, e.g. SACK. Here, if SACK were placed as an Inner Option, another 'ISN' option would be needed to detect and allow for the ISN shift.

Flow-control deadlock: It needs to be proved whether the solution to flow-control deadlock for acknowledgement-related options also avoids the risk of deadlock across one or more connection-splitting middleboxes.

Simultaneous open: If host A sends a SYN-U from port S to D, it might receive a SYN rather than a SYN/ACK on port S from port D. Whether the SYN is upgraded or not, it is believed that it will be possible to define all the cases necessary to fully specify the simultaneous open case. The number of combinations that have to be considered becomes quite tiresome, especially if the case of simulataneous dual handshakes is included. Therefore, these corner-cases will be addressed in a later revision.

TCP offload: The protocol design is intended to ensure that new TCP extensions will survive segmentation offload. The InSpace Options are also intended to provide a robust way for an Inner Space TCP

to offload the generation or ingestion of TCP segments without breaking extensibility, but whether it is the best way to interwork with offload hardware is yet to be determined.

Appendix F. 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-01 to briscoe-...-inner-space-sink-00:
Technical changes:

* Added choice of in-order and out-of-order TCP option delivery

* Added padding for 4-octet alignment of options

* Made InSpace Options for SYN=0 or SYN=1 have the same structure by i) including magic no / message boundary marker as prefix to InSpace option and ii) allowing Prefix (out-of-order or fire-and-forget) Options in all segments.

* Changed Sent Payload Size (SPS) field to Sent Data Size (SDS), to minimise framing arithmetic.

* Allowed space in the InSpace Option for the SOO field on all segments (not just SYN=1). Also allowed a choice of Len=1 or 2 when SYN=0 and introduced the P flag if Len=1 to state whether the Inner Options are all Prefix or all Suffix.

* Added the Marker and ZOMBI fields to the InSpace Option when SYN=0.

* Extended Sequence Space Consumption rules to require the sequence space of fire-and-forget objects to be coinsidered implicitly acknowledged.

* Removed Fire-and-Forget Options from flow control coverage.

* New rules for new concept of Impure ACKs.

* Defined Construction Order for writing TCP Data.

* Extensive changes to processing order when reading Inner Options with SYN=0.

* 'Compatibility with Pre-Existing TCP Variants' now categorises existing TCP options by whether they must be Prefix, Suffix or

either, and requires future option definitions to make this distinction. Also added some previously overlooked options (no-op & EOL) and re-categorised TCP-AO, with explanation

* When explicit port binding needed, recommended dual handshakes in series rather than disabling Inner Space.

* Defined behaviour when app attempts to determine PMTU.

* Added security recommendation not to block data-in-SYN unless other signs of SYN flood attack.

* Discussed the potential new attack vector in the optional DPI traversal approach, and why it is probably not a concern now that the approach is only used in the client-server direction.

* Made ModeSwitch mandatory, not optional.

* Restructured the InSpace Option for a jumbogram

* Specified that the optional DPI traversal extension would only be used in the client-server direction, and restructured to remain consistent with the changes to the regular InSpace Option structure.

* Cleared all Protocol Design Issues, and added some new ones.

Editorial changes:

* Changes to document structure:

  + Added Wider Implications subsection to Intro, looking forward to i) a structured control channel for end-to-middle interaction and ii) new transport services such as Multiplexed streams, compression and encryption;

  + Added 'Flow Control Coverage' and 'Construction Order for TCP Data' subsections to 'Writing Inner TCP Options' section;

  + Added 'Header Extension by Encapsulation' and 'Framing Segments' subsections to rationale for Inner Option Space;

  + Split 'Control Options Within Data Sequence Space' into two subsections: i) 'In-Order Flow-Controlled Options' using the existing text and a new 'Fire-and-Forget Options' subsection;

  + Added 'Deployment Approach', including 'Substrate Protocol: TCP vs. UDP', and ''User-Space vs. Kernel-Space' to Rationale section;

  + Promoted Protocol Overhead subsection.

  + Added appendix for 'Zero Overhead Message Boundary Insertion (ZOMBI)';

* Abstract & Introduction: primary goal changed to redesign of TCP's extensibility mechanism (ie middlebox traversal as well as option space).

* Introduction:

  + Rewrote Introduction to introduce the two difficult questions that tunnelling TCP options raises: i) immediate (out-of-order) delivery of certain options and ii) bootstrapping the inner control channel;

  + Made examples in Intro consistent with those in TCP Compatibility section (i.e. TCP-AO removed from Inner Option list).

  + Added MPTCP & tcpinc to 'Motivation for Adoption Now'

* Terminology: Added definitions of Pure ACKs, Impure ACKs and Flow-Controlled ACKs.

* Protocol Spec

  + Upgraded Segment Structure and Format: Reflected technical changes as above

  + Inner TCP Option Processing: Introduced distinction between flow-controlled and fire-and-forget options at the start

* Acknowledged more helpful people.

* Added refs related to Minion/COBS, HTTP2 and an architectural paper on Inner Space.

* Appendices: Expanded rationale for optional DPI traversal fall-back if not supported by both ends.

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 previous 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/