Internet Engineering Task Force	A. Ford
Internet-Draft	Roke Manor Research
Intended status: Experimental	C. Raiciu
Expires: September 10, 2010	M. Handley
	University College London
	March 09, 2010

TOC

TCP Extensions for Multipath Operation with Multiple Addresses
draft-ford-mptcp-multiaddressed-03

Abstract

TCP/IP communication is currently restricted to a single path per connection, yet multiple paths often exist between peers. The simultaneous use of these multiple paths for a TCP/IP session would improve resource usage within the network, and thus improve user experience through higher throughput and improved resilience to network failure.
Multipath TCP provides the ability to simultaneously use multiple paths between peers. This document presents a set of extensions to traditional TCP to support multipath operation. The protocol offers the same type of service to applications as TCP - reliable bytestream - and provides the components necessary to establish and use multiple TCP flows across potentially disjoint paths.

Status of this Memo

Table of Contents

---

1. Introduction

TOC

Multipath TCP (henceforth referred to as MPTCP) is set of extensions to regular TCP [2] (Postel, J., “Transmission Control Protocol,” September 1981.) to allow a transport connection to operate across multiple paths simultaneously. This document presents the protocol changes required by Multipath TCP, specifically those for signalling and setting up multiple paths ("subflows"), managing these subflows, reassembly of data, and termination of sessions. This is not the only information required to create a Multipath TCP implementation, however. This document is complemented by several others:

- Architecture [3] (Ford, A., Raiciu, C., Barre, S., and J. Iyengar, “Architectural Guidelines for Multipath TCP Development,” March 2010.), which explains the motivations behind Multipath TCP and a functional separation through which an extensible MPTCP implementation can be developed.

- Congestion Control [4] (Raiciu, C., Handley, M., and D. Wischik, “Coupled Multipath-Aware Congestion Control,” October 2009.), presenting a safe congestion control algorithm for coupling the behaviour of the multiple paths in order to "do no harm" to other network users.

- Application Considerations [5] (Scharf, M. and A. Ford, “MPTCP Application Interface Considerations,” October 2009.), discussing what impact MPTCP will have on applications, what applications will want to do with MPTCP, and as a consequence of these factors, what API extensions an MPTCP implementation should present.

TOC

1.1. Design Assumptions

In order to limit the potentially huge design space, the authors imposed two key constraints on the multipath TCP design presented in this document:

- It must be backwards-compatible with current, regular TCP, to increase its chances of deployment
- It can be assumed that one or both endpoints are multihomed and multiaddressed

To simplify the design we assume that the presence of multiple addresses at an endpoint is sufficient to indicate the existence of multiple paths. These paths need not be entirely disjoint: they may share one or many routers between them. Even in such a situation making use of multiple paths is beneficial, improving resource utilisation and resilience to a subset of node failures. The congestion control algorithms as discussed in [4] (Raiciu, C., Handley, M., and D. Wischik, “Coupled Multipath-Aware Congestion Control,” October 2009.) ensure this does not act detrimentally.
There are three aspects to the backwards-compatibility listed above (discussed in more detail in [3] (Ford, A., Raiciu, C., Barre, S., and J. Iyengar, “Architectural Guidelines for Multipath TCP Development,” March 2010.)):

> **External Constraints:** The protocol must function through the vast majority of existing middleboxes such as NATs, firewalls and proxies, and as such must resemble existing TCP as far as possible on the wire. Furthermore, the protocol must not assume the segments it sends on the wire arrive unmodified at the destination: they may be split or coalesced; options may be removed or duplicated.
>
> **Application Constraints:** The protocol must be usable with no change to existing applications that use the standard TCP API (although it is reasonable that not all features would be available to such legacy applications).
>
> **Fall-back:** The protocol should be able to fall back to standard TCP with no interference from the user, to be able to communicate with legacy hosts.

Areas for further study:

- In theory, since this is purely a TCP extension, it should be possible to use MPTCP with both IPv4 and IPv6 subflows for the same connection on dual-stack hosts, thus having the additional possible benefit of aiding transition.

*Some features of the design presented here could be extended to work with non-multi-addressed hosts by using other packet metadata (such as ports or flow label), packet marking, or partial multipath (such as by using a proxy).

1.2. Layered Representation

TOC

MPTCP operates at the transport layer, and its existence aims to be transparent to both higher and lower layers. It is a set of additional features on top of standard TCP, and as such MPTCP is designed to be usable by legacy applications with no changes. A possible implementation would be for such a feature to be a system-wide setting: "Use multipath TCP by default? Y/N". Multipath-aware applications would be able to use an extended sockets API to have further influence on the behaviour of MPTCP. Figure 1 (Comparison of Standard TCP and MPTCP Protocol Stacks) illustrates this layering.

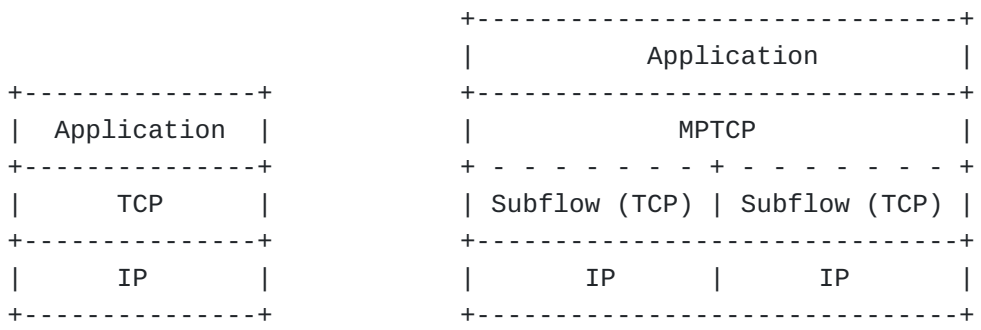

Figure 1: Comparison of Standard TCP and MPTCP Protocol Stacks

---

Detailed discussion of an architecture for developing a multipath TCP implementation, especially regarding the functional separation by which different components should be developed, is given in [3] (Ford, A., Raiciu, C., Barre, S., and J. Iyengar, “Architectural Guidelines for Multipath TCP Development,” March 2010.).

1.3. Operation Summary

TOC

This section provides a high-level summary of normal operation of MPTCP, and is illustrated by the scenario shown in Figure 2 (Example

MPTCP Usage Scenario). A detailed description of operation is given in Section 4 (MPTCP Protocol).

*To a non-MPTCP-aware application, MPTCP will be indistinguishable from normal TCP. All MPTCP operation is handled by the MPTCP implementation, although extended APIs could provide additional control and influence [5] (Scharf, M. and A. Ford, “MPTCP Application Interface Considerations,” October 2009.). An application begins by opening a TCP socket in the normal way.

*An MPTCP connection begins as a single TCP session. This is illustrated in Figure 2 (Example MPTCP Usage Scenario) as being between Addresses A1 and B1 on Hosts A and B respectively.

*If extra paths are available, additional TCP sessions are created on these paths, and are combined with the existing session, which continues to appear as a single connection to the applications at both ends. The creation of the additional TCP session is illustrated between Address A2 on Host A and Address B1 on Host B.

*MPTCP identifies multiple paths by the presence of multiple addresses at endpoints. Combinations of these multiple addresses equate to the additional paths. In the example, other potential paths that could be set up are A1<->B2 and A2<->B2. Although this additional session is shown as being initiated from A2, it could equally have been initiated from B1.

*The discovery and setup of additional TCP sessions (termed 'subflows') will be achieved through a path management method. This document describes a mechanism by which an endpoint can initiate new subflows by using its additional addresses, or by signalling its available addresses to the other endpoint.

*MPTCP adds connection-level sequence numbers to allow the reassembly of the in-order data stream from multiple subflows which may deliver packets out-of-order due to differing network delays. Connections are terminated by connection-level FIN packets as well as those relating to the individual subflows.

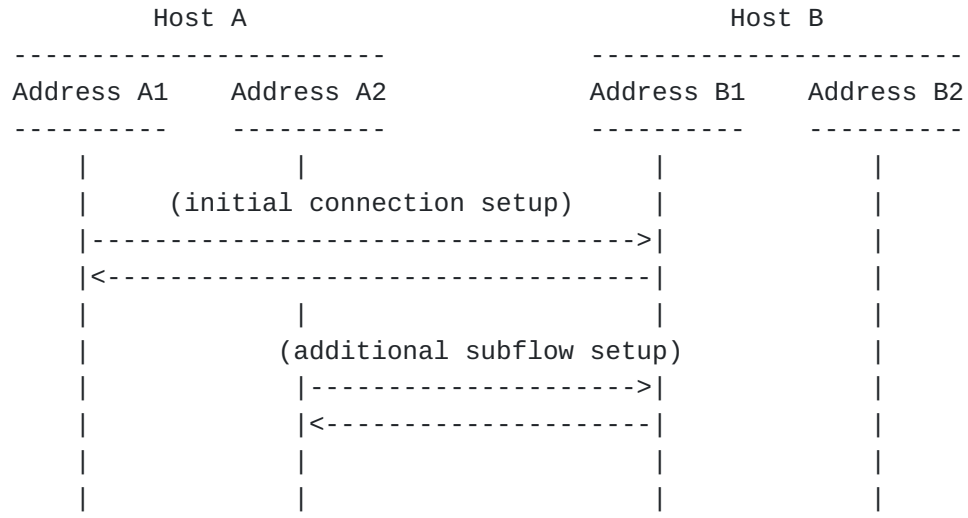

Figure 2: Example MPTCP Usage Scenario

1.4. Requirements Language

TOC

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 (Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels," March 1997.) [1].

2. Terminology

TOC

Path: A sequence of links between a sender and a receiver, defined in this context by a source and destination address pair.

Subflow: A stream of TCP packets sent over a path. A subflow is a component part of a connection between two endpoints.

Connection: A collection of one or more subflows, over which an application can communicate between two endpoints. There is a one-to-one mapping between a connection and a socket.

Data-level: The payload data is nominally transfered over a connection, which in turn is transported over subflows. Thus the term "data-level" is synonymous with "connection level", in

contrast to "subflow-level" which refers to properties of an individual subflow.

Token: A locally unique identifier given to a multipath connection by an endpoint. May also be referred to as a "Connection ID".

Endpoint: A host operating an MPTCP implementation, and either initiating or terminating a MPTCP connection.

---

3. Semantic Issues

TOC

In order to support multipath operation, the semantics of some TCP components have changed. To aid clarity, this section collects these semantic changes as a reference.

Sequence Number: The (in-header) TCP sequence number is specific to the subflow. To allow the receiver to reorder application data, an additional data-level sequence space is used. In this data-level sequence space, the initial SYN and the final DATA_FIN occupy one octet. There is an explicit mapping of data sequence space to subflow sequence space, which is signalled through TCP options in data packets.

ACK: The ACK field in the TCP header acknowledges the subflow sequence number only, not the data-level sequence space. Although data acknowledgments could be inferred from the subflow ACK, an explicit connection-level DATA_ACK is used to ensure end-to-end reliability in the presense of certain types of middlebox.

Receive Window: The receive window in the TCP header indicates the amount of free buffer space for this connection (as opposed to for this subflow) that is available at the receiver. This is a change to the semantics of the field. With regular TCP the window is relative to the acknowledgment number in the TCP header. This is not meaningful for multipath TCP. Instead with multipath TCP the receive window is relative to the DATA_ACK field, indicating the amount of buffer space available at the data-level. This permits the receive window to serve its original purpose and

provide flow-control of the data sent by the TCP sending application.

FIN: The FIN flag in the TCP header applies only to the subflow it is sent on, not to the whole connection. For connection-level FIN semantics, the DATA_FIN option is used.

RST: The RST flag in the TCP header applies only to the subflow it is sent on, not to the whole connection. A connection is considered reset if a RST is received on every subflow.

Address List: Address management is handled on a per-connection basis (as opposed to per-subflow, per host, or per pair of communicating hosts). This permits the application of per-connection local policy. Adding an address to one connection has no implication whatsoever for other connections between the same pair of hosts.

5-tuple: The 5-tuple (protocol, local address, local port, remote address, remote port) presented to the application layer in a non-multipath-aware application is that of the first subflow, even if the subflow has since been closed and removed from the connection. These API issues are discussed in more detail in [5] (Scharf, M. and A. Ford, "MPTCP Application Interface Considerations," October 2009.).

---

4. MPTCP Protocol

TOC

This section describes the operation of the MPTCP protocol, and is subdivided into sections for each key part of the protocol operation. All MPTCP operations are signalled using optional TCP header fields. These TCP Options will have option numbers allocated by IANA, as listed in Section 10 (IANA Considerations), and are defined throughout the following subsections.

4.1. Connection Initiation

TOC

Connection Initiation begins with a SYN, SYN/ACK exchange on a single path. Each of these packets will additionally feature the MP_CAPABLE TCP option (Figure 3 (Multipath Capable option)) This option declares its sender is capable of performing multipath TCP and wishes to do so on this particular connection). As well as this declaration, this field

presents a locally-unique token identifying this connection. This is used when adding additional subflows to this connection.
This token is generated by the sender and has local meaning only, hence it MUST be unique for the sender. The token MUST be difficult for an attacker to guess, and thus it is recommended it SHOULD be generated randomly. (However, see further discussions about security in Section 5 (Security Considerations), including the possibility of 64-bit tokens.)
This option is only present in packets with the SYN flag set. It is only used in the first TCP session of a connection, in order to identify the connection; all following connections will use path management options (see Section 4.2 (Starting a New Subflow)) to join the existing connection.

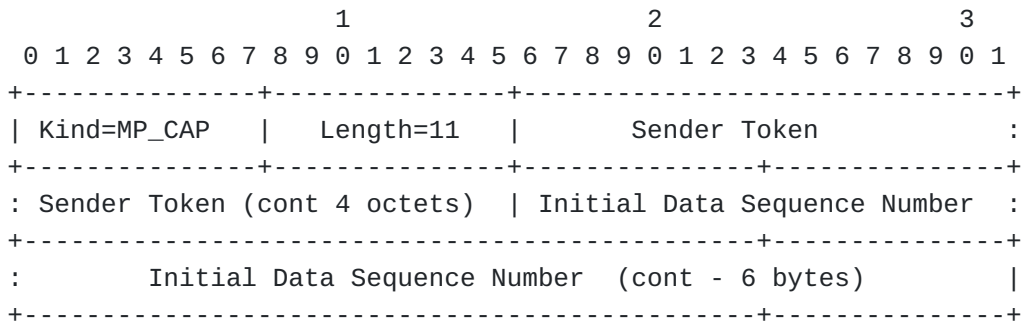

Figure 3: Multipath Capable option

If a SYN contains an MP_CAPABLE option but the SYN/ACK does not, it is assumed that the passive opener is not multipath capable and thus the MPTCP session will operate as regular, single-path TCP. If a SYN does not contain a MP_CAPABLE option, the SYN/ACK MUST NOT contain one in response.
If the SYN packets are unacknowledged, it is up to local policy to decide how to respond. It is expected that a sender will eventually fall back to single-path TCP (i.e. without the MP_CAPABLE Option), in order to work around middleboxes that may drop packets with unknown options; however, the number of multipath-capable attempts that are made first will be up to local policy. Once the active opener has sent a SYN without the MP_CAPABLE option, it MUST fall back to regular TCP behavior, even if it subsequently receives a SYN/ACK that contains an MP_CAPABLE option. This might happen if the MP_CAPABLE SYN and subsequent non-MP-capable SYN are reordered. This is to ensure that the two endpoints end up in an interoperable state, no matter what order the SYNs arrive at the passive opener. This final state is inferred

from the presence or absence of the DATA_ACK option in the third packet of the TCP handshake.
The MPC option includes the most significant 6 bytes of the 8-byte initial Data Sequence Number option (discussed in Section 4.4 (General MPTCP Operation)). The least significant two bytes should be zeroed. This is also used as an implicit mapping of the SYN to the data sequence space (and this initial SYN counts as one octet in this space, as for a regular SYN in single-path TCP). This will be used to ensure both ends agree on whether the connection is multipath or standard TCP, regardless of middlebox behaviour. This could also have some (minor) security benefits, discussed in Section 5 (Security Considerations). To preserve option space, only the most significant six bytes are sent in the SYN, as there is no significant security benefit from randomizing the values of the lower two bytes given that these fall within typical receive windows sizes.

4.2. Starting a New Subflow

TOC

Endpoints have knowledge of their own address(es), and can become aware of the other endpoint's addresses through signalling exchanges as described in Section 4.3 (Address Knowledge Exchange (Path Management)). Using this knowledge, an endpoint can initiate a new subflow over a currently unused pair of addresses. Either endpoint that is part of a connection can initiate the creation of a new subflow.
A new subflow is started as a normal TCP SYN/ACK exchange. The "Join" TCP option (Figure 4 (Join Connection option)) is used to identify of which connection the new subflow should become a part. The token used is the locally unique token of the destination for the subflow, as defined by the MP_CAPABLE option received in the first SYN/ACK exchange.

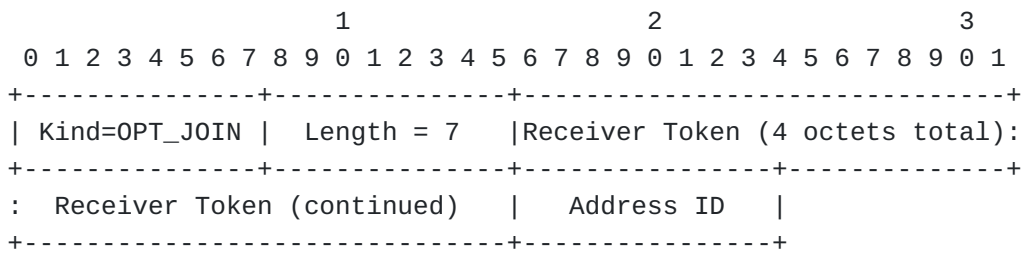

Figure 4: Join Connection option

In response to a SYN with the "Join" option, if the token is valid for an existing MPTCP connection, the recipient MUST respond with a SYN/ACK also containing a "Join" option, with the initiator's token. This serves two purposes: firstly, to ensure both endpoints agree on the connection being referred to (this is particularly relevant when both addresses being used are new to the connection); and secondly, to ensure there are no middleboxes in the path that will drop MPTCP options on the return path. This behaviour is illustred in Figure 5 (Example use of MPTCP Tokens).

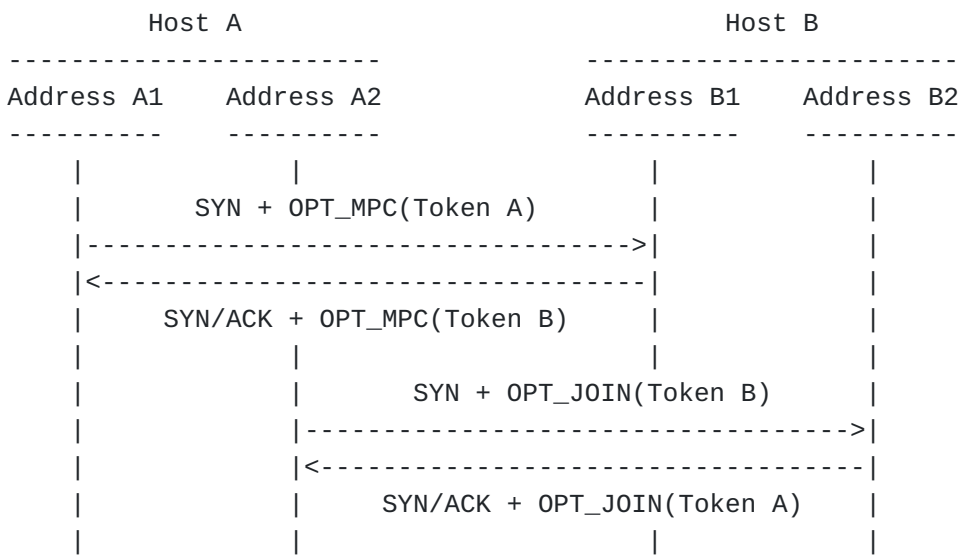

Figure 5: Example use of MPTCP Tokens

If the token is unknown, the recipient MUST respond with a TCP RST in the same way as when an unknown TCP port is used.
It should be noted that additional subflows can exist between any pair of ports; no explicit accept calls or bind calls are required to open additional subflows. To associate a new subflow to an existing connection, the token supplied in the subflow's SYN exchange is used for demultiplexing. This means that port numbers on subflow SYN exchanges are not important, and a receiver of a SYN SHOULD allow any values to be used, as long as the 5-tuple is unique for each host. However the sender of a SYN containing a JOIN option SHOULD send the SYN to the port used by the remote party for the first subflow in the connection. The local port for such SYNs MAY be chosen locally, either dynamically, or by the application if an API allows the application to do so. This strategy is intended to maximize the probability of the SYN being permitted by a firewall or NAT at the recipient and to avoid confusing any network monitoring software.

Deumultiplexing subflow SYNs MUST be done using the token; this is unlike traditional TCP, where the destination port is used for demultiplexing SYN packets. Once a subflow is setup, demultiplexing packets is done using the five-tuple, as in traditional TCP.
The JOIN option includes an "Address ID". This is an identifier, locally unique to the sender of this option, and with only per-connection relevance, which identifies the source address of this packet. The key purpose of this identifier is, if an address becomes unexpectedly unavailable on the sender, it can signal this to the receiver via a remove address option (Section 4.3.2 (Remove Address)) without needing to know what the source address actually is (thus allowing the use of NATs). It also allows correlation between new connection attempts and address signalling (Section 4.3.1 (Address Advertisement)), to prevent duplicate subflow initiation.
The Address IDs of the subflow used in the initial SYN exchange of the first subflow in the connection are implicit, and have the value zero. The Address ID must be stored by the receiver in a data structure that gathers all the Address ID to address mappings for a connection identified by a token pair. In this way there is a stored mapping between Address ID, observed source address and token pair for future processing of control information for a connection.
This option can only be present when the SYN flag is set.

4.3. Address Knowledge Exchange (Path Management)

TOC

We use the term "path management" to refer to the exchange of information about additional paths between endpoints, which in this design is managed by multiple addresses at endpoints. For more detail of the architectural thinking behind this design, see the separate document [3] (Ford, A., Raiciu, C., Barre, S., and J. Iyengar, “Architectural Guidelines for Multipath TCP Development,” March 2010.). This design makes use of two methods of sharing such information, used simultaneously. The first is the direct setup of new subflows, already described in Section 4.2 (Starting a New Subflow), where the initiator has an additional address. The second method is described in the following subsections, whereby addresses are signalled explicitly to the other endpoint, to allow it to initiate new connections. This approach, of two complementary mechanisms, has been chosen to allow addresses to change in flight, and thus support operation through NATs, whilst also allowing the signalling of previously unknown addresses, such as those belonging to other address families (e.g. IPv4 and IPv6). Here is an example of typical operation of the protocol:

*An endpoint that is multihomed starts an additional TCP session to an address/port pair that is already in use on the other endpoint, using a token to identify the flow (Section 4.2

(Starting a New Subflow). (A multihomed destination may open a new subflow from its new address to an existing subflow's source address and port, or a multihomed source may open a new subflow from its new address to an existing subflow's destination and port).

*More concretely, say a connection is intiated from host "A" on (address, port) combination A1 to destination (address, port) B1 on host "B". If host A is multihomed, it starts an additional connection from new (address, port) A2 to B1, using B's previously declared token. Alternatively, if B is multhomed, it will try to set up a new TCP connection from B2 to A1, using A's previously declared token.

*Simultaneously (or after a timeout), an "Add Address" option (Section 4.3.1 (Address Advertisement)) is sent on an existing subflow, informing the receiver of the sender's alternative address(es). The recipient can use this information to open a new subflow to the sender's additional address. Using the previous notation, this would be an Add Address packet sent from A1 to B1, informing B of address A2.

*The mix of using the SYN-based option and the Add Address option, including timeouts, is implementation-specific and can be tailored to agree with local policy.

*If host B successfully receives the first SYN, starting a new subflow, it can use the Address ID in the Join option to correlate this with the Add Address option that will also arrive on an existing subflow. Assuming the endpoint has already responded to the SYN with a SYN/ACK, it will know to ignore the Add Address option. Otherwise, if it has not received such a SYN, it will try to initiate a new subflow from one or more of its addresses to address A2 (triggered by the Add Address option). This is intended to permit new sessions to be opened if one endpoint is behind a NAT. A slight security improvement can be gained if a host ensures there is a correlated Add Address option before responding to the SYN.

Other scenarios are valid, however, such as those where entirely new addresses are signalled, e.g. to allow an IPv6 and an IPv4 path to be used simultaneously.

4.3.1. Address Advertisement

TOC

The Add Address TCP Option announces additional addresses on which an endpoint can be reached (Figure 6 (Add Address option (for IPv4)),

which allows several (ID, address) pairs to be announced to the other endpoint. Multiple addresses can be added if there is sufficient TCP option space, otherwise multiple TCP messages containing this option will be sent. This option can be used at any time during a connection, depending on when the sender wishes to enable multiple paths and/or when paths become available.

Every address has an ID which can be used for address removal, and therefore endpoints must cache the mapping between ID and address. This is also used to identify Join Connection options (Section 4.2 (Starting a New Subflow)) relating to the same address, even when address translators are in use. The ID must be unique to the sender and connection, per address, but its mechanism for allocating such IDs is implementation-specific.

This option is shown for IPv4. For IPv6, the IPVer field will read 6, and the length of the address will be 16 octets not 4, and thus the length of the option will be 2 + (18 * number_of_entries). If there is sufficient TCP option space, multiple addresses can be included, with an ID following on immediately from the previous address, and their existance can be inferred through the option length and version fields.

NB: by having a IPVer field, we get four free reserved bits. These could be used in later versions of this protocol for expressing sender policy, e.g. one bit for "use now" or similar, to differentiate between subflows for backup purposes and those for throughput.

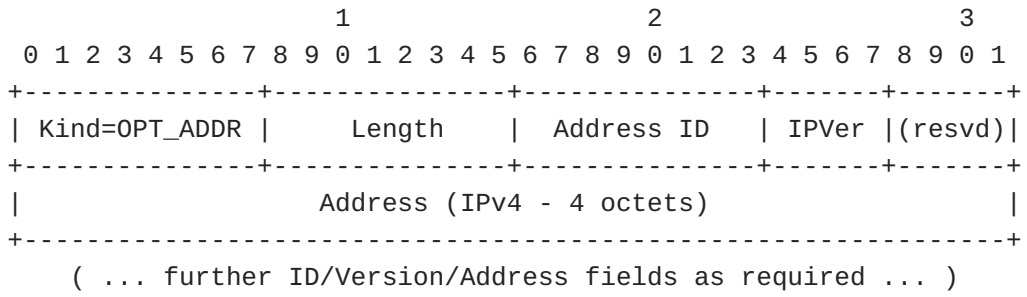

Figure 6: Add Address option (for IPv4)

Ideally, we'd like to ensure the Add Address (and Remove Address) option is sent reliably and in order to the other end. This is to ensure that we don't close the connection when remove/add addresses are processed in reverse order, and to ensure that all possible paths are used. We note, however, that losing reliability and ordering it will not break the multipath connections; they will just reduce the opportunity to open multipath paths and to survive different patterns of path failures.

Subflow level ACKs do not cover options, so if we want explicit guarantees we need to build in other mechanisms. Solutions include

echoing the options and sending one option per RTT, or adding a sequence number to the option which is explicitly acked in another option. However, we feel these mechanisms' added complexity is not worth the benefits they bring. There are two basic failure modes for options: a) every new option gets stripped or b) some options get stripped, randomly. The second option looks more like a middlebox implementation error, so we believe it is not worth optimizing for. In the first case, resending the option on a different subflow is the thing to do. To achieve similar reliability without explicit ACKs, we propose sending all Add/Remove Address options on all existing subflows. If ordering is needed, we should only send one add/remove option per RTT (modulo lost packets at subflow level).
If an address index is in use, the Add Address option SHOULD be silently ignored.

4.3.2. Remove Address

TOC

If, during the lifetime of a MPTCP connection, a previously-announced address becomes invalid (e.g. if the interface disappears), the affected endpoint should announce this so that the other endpoint can remove subflows related to this address.
This is achieved through the Remove Address option (Figure 7 (Remove Address option)), which will remove a previously-added address (or list of addresses) from a connection and terminate any subflows currently using that address.
The sending and receipt of this message should trigger the sending of FINs by both endpoints on the affected subflow(s) (if possible), as a courtesy to cleaning up middlebox state, but endpoints may clean up their internal state without a long timeout.
Address removal is undertaken by ID, so as to permit the use of NATs and other middleboxes. If there is no address at the requested ID, the receiver will silently ignore the request.
The standard way to close a subflow (so long as it is still functioning) is to use a FIN exchange as in regular TCP - for more information, see Section 4.5 (Closing a Connection).

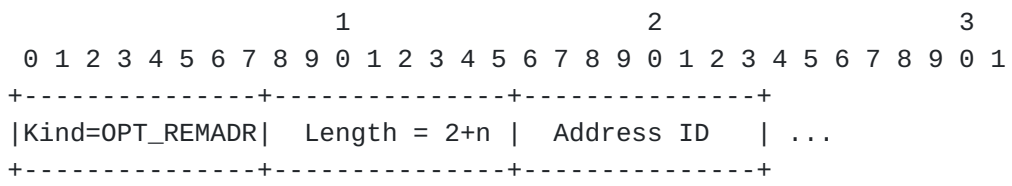

Figure 7: Remove Address option

4.4. General MPTCP Operation

TOC

This section discusses operation of MPTCP for data transfer. At a high level, an MPTCP implementation will take one input data stream from an application, and split it into one or more subflows, with sufficient control information to allow it to be reassembled and delivered reliably and in-order to the recipient application. The following subsections define this behaviour in detail.

4.4.1. Data Sequence Numbering

TOC

The data stream as a whole can be reassembled through the use of the Data Sequence Mapping (Figure 8 (Data Sequence Mapping option)) option, which defines the mapping from the data sequence number to the subflow sequence number. This is used by the receiver to ensure in-order delivery to the application layer. Meanwhile, the subflow-level sequence numbers (i.e. the regular sequence numbers in the TCP header) have subflow-only relevance. It is expected (but not mandated) that SACK [6] (Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, "TCP Selective Acknowledgment Options," October 1996.) is used at the subflow level to improve efficiency.

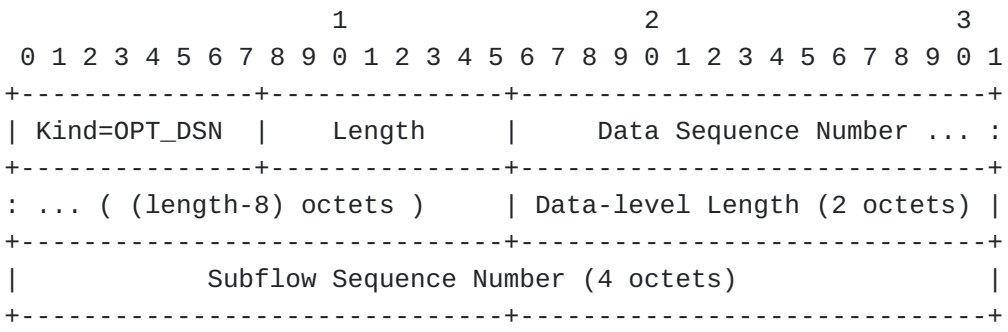

Figure 8: Data Sequence Mapping option

This option specifies a full mapping from data sequence number to subflow sequence number, informing the receiver that there is a one-to-one correspondence between the two sequence spaces for the specified length. The purpose of the explicit mapping is to assist with

compatibility with situations where TCP/IP segmentation or coalescing is undertaken separately from the stack that is generating the data flow (e.g. through the use of TCP segmentation offloading on network interface cards, or by middleboxes such as performance enhancing proxies).

The data sequence number specified in this option is absolute, whereas the subflow sequence numbering is relative (the SYN at the start of the subflow has subflow sequence number 1). This is to permit middleboxes that may wish to alter sequence numbering, since the data stream itself will not be affected.

TBD: if we used absolute sequence numbers that would make receiver code a bit simpler, and would make it more difficult to inject data as the attacker needs to guess both Data Sequence Number and Subflow Sequence Number. How many middleboxes are there that change the sequence numbers, and should we optimize for them?

A mapping is unique, in that the subflow sequence number is bound to the data sequence number after the mapping has been processed. It is not possible to change this mapping afterwards; however, the same data sequence number can be mapped on different subflows for retransmission purposes (see Section 4.4.4 (Sender Considerations)).

A receiver MUST NOT accept data for which it does not have a mapping to the data sequence space. To do this, the receiver will not acknowledge the unmapped data at subflow level. It is better to have a subflow fail than to accept data in the wrong order. However, if there was a lost packet in the subflow, the receiver SHOULD wait for this to be retransmitted before closing the subflow, since the lost packet may contain the necessary mapping information.

NOTE: if the subflow did ACK data for which it did not have a mapping, it would be possible to use the DATA_ACK to detect when the mapping was lost. This will likely not increase reliability, as the subflow will likely drop all unknown options. In addition, the receiver is now storing potentially useless data: what happens if the mapping never arrives? Should the receiver have a timer to delete this data?

Data sequence numbers are always 64-bit quantities, and should be maintained as such in implementations. If a connection is progressing at a slow rate, so that protection against wrapped sequence numbers is not required, and security requirements against blind insertion attacks are not stringent, then it is permissible to include just the lower 32 bits of the sequence number in the OPT_DSN option as an optimization. Implementations MUST accept this and implicitly promote it to a 64-bit quantity. In all other cases, the full 64 bits should be included. Security implications are discussed in Section 5 (Security Considerations).

As with the standard TCP sequence number, the data sequence number should not start at zero, but at a random value to make session hijacking harder. This is done by including a Data Sequence Mapping option along with the MP_CAPABLE option in the initial SYN (which occupies one octet of data sequence space; see Section 4.1 (Connection Initiation)). In this case, to save option space, neither the data-

level length nor the subflow sequence number fields are present in this option, so the Length field will be the length of the Data Sequence Number, plus two octets.
The Data Sequence Mapping does not need to be included in every MPTCP packet, as long as the subflow sequence space in that packet is covered by a mapping known at a receiver. This can be used to reduce overhead in cases where the mapping is known in advance; one such case is when there is a single subflow between the endpoints, another is when segments of data are scheduled in larger than packet-sized chunks.

4.4.2. Data Acknowledgements

TOC

In a perfect world, it would be possible to make do with only subflow-level acknowledgements, with the sender keeping track of these acknowledgements to derive what data has been successfully received. If there are ever cases where the subflow data is dropped after it has been acked (which may occur if a proxy middlebox fails, or if a buffer fills on a host), the connection will break entirely since the sender will assume the data has been received when it hasn't.
Therefore, MPTCP provides a connection-level acknowledgement (the DATA_ACK) to act as a cumulative ACK for the connection as a whole. This is analogous to the behaviour of the standard TCP cumulative ACK in SACK - indicating how much data has been successfully received (with no holes). This option, illustrated in Figure 9 (Connection-level Acknowledgement (DATA_ACK)), is expected to be included in every packet by an MPTCP host.

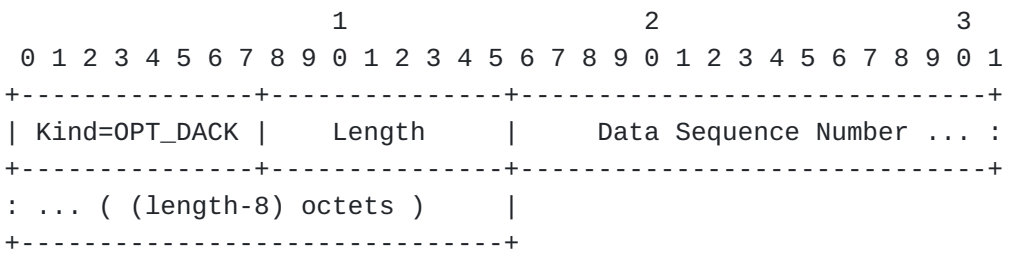

Figure 9: Connection-level Acknowledgement (DATA_ACK)

TOC

4.4.3. Receiver Considerations

Regular TCP advertises a receive window in each packet, telling the sender how much data the receiver is willing to accept past the cumulative ack. The receive window is used to implement flow control, throttling down fast senders when receivers cannot keep up.
MPTCP also uses a unique receive window, shared between the subflows. The idea is to allow any subflow to send data as long as the receiver is willing to accept it; the alternative, maintaining per subflow receive windows, could end-up stalling some subflows while others would not use up their window.
The receive window is relative to the DATA_ACK. As in TCP, a receiver MUST NOT shrink the right edge of the receive window (e.g. DATA_ACK + receive window). The receiver will use the Data Sequence Number to tell if a packet should be accepted at connection level.
When deciding to accept packets at subflow level, normal TCP uses the sequence number in the packet and checks it against the allowed receive window. With multipath, such a check is done using only the connection level window. A sanity check could be performed at subflow level to ensure that: SSN-SUBFLOW_ACK <= DSN - DATA_ACK.
When should segments be processed at connection level? The default is to wait until they arrive in order at subflow level, and only then do connection level processing. However, one can optimize for speed by processing at connection level segments that have not yet been acked at subflow level; the only requirement for this optimization is to have a valid data sequence mapping for the segment. Note that the segment can be dropped at subflow level afterwards (e.g. because it is out of order and there is more pressure); the DATA_ACK ensure the connection can make progress without having to wait for the subflow retransmission.
An issue will arise regarding how large a receive buffer to implement. The lower bound would be the maximum bandwidth/delay product of all paths, however this could easily fill when a packet is lost on a slower subflow and needs to be retransmitted (see Section 4.4.4 (Sender Considerations)). The upper bound would be the maximum RTT multiplied by the maximum total bandwidth available. This will cover most eventualities, but could easily become very large. It is FFS what the best approach is.

4.4.4. Sender Considerations

TOC

The sender should only consider receive window advertisements where the largest sequence number allowed (i.e. DATA_ACK + receive window) increases. This is important to allow using paths with different RTTs, and thus different feedback loops.
The data sequence mapping allows senders to re-send data with the same data sequence number on a different subflow. When doing this, an

endpoint must still retransmit the original data on the original subflow, in order to preserve the subflow integrity (middleboxes could replay old data, and/or could reject holes in subflows), and a receiver will ignore these retransmissions. While this is clearly suboptimal, for compatibility reasons this is the best behaviour. Optimisations could be negotiated in future versions of this protocol.
This protocol specification does not mandate any mechanisms for handling retransmissions, and much will be dependent upon local policy (as discussed in Section 4.4.6 (Subflow Policy)). One can imagine aggressive connection level retransmissions policies where every packet lost at subflow level is retransmitted on a different subflow (hence wasting bandwidth but possibly reducing application-to-application delays), or conservative retransmission policies where connection-level retransmits are only used after a few subflow level retransmission timeouts occur.
Whichever the retransmission strategy, the sender MUST keep data in its send buffer as long as the data has not been acked at connection level and on all subflows it has been sent on. In this way, the sender can always retransmit the data if needed, on the same subflow or on a different one. A special case is when a subflow fails: the sender will typically resend the data on other working subflows, and will keep trying to retransmit the data on the failed subflow too. The sender will declare the subflow failed after a predefined upper bound on retransmissions is reached, and only then delete the outstanding data segments.
A sender will maintain connection level timers for unacknowledged segments. These timers will be based on the subflow timers, and will guard against pro-active acking by middleboxes.
The send buffer must be, at the minimum, as big as the receive buffer, to enable the sender to reach maximum throughput.

4.4.5. Congestion Control Considerations

TOC

Different subflows in an MPTCP connection have different congestion windows. To achieve resource pooling, it is necessary to couple the congestion windows in use on each subflow, in order to push most traffic to uncongested links. One algorithm for achieving this is presented in [4] (Raiciu, C., Handley, M., and D. Wischik, “Coupled Multipath-Aware Congestion Control,” October 2009.); the algorithm does not achieve perfect resource pooling but is "safe" in that it is readily deployable in the current Internet.
It is foreseeable that different congestion controllers will be implemented for MPTCP, each aiming to achieve different properties in the resource pooling/fairness/stability design space. Much research is expected in this area in the near future.

Regardless of the algorithm used, the design of the MPTCP protocol aims to provide the congestion control implementations sufficient information to take the right decisions; this information includes, for each subflow, which packets where lost and when.

4.4.6. Subflow Policy

TOC

Within a local MPTCP implementation, a host may use any local policy it wishes to decide how to share the traffic to be sent over the available paths.
In the typical use case, where the goal is to maximise throughput, all available paths will be used simultaneously for data transfer, using coupled congestion control as described in [4] (Raiciu, C., Handley, M., and D. Wischik, “Coupled Multipath-Aware Congestion Control,” October 2009.). It is expected, however, that other use cases will appear.
For instance, a possibility is an 'all-or-nothing' approach, i.e. have a second path ready for use in the event of failure of the first path, but alternatives could include entirely saturating one path before using an additional path (the 'overflow' case). Such choices would be most likely based on the monetary cost of links, but may also be based on properties such as the delay or jitter of links, where stability is more important than throughput. Application requirements such as these are discussed in detail in [5] (Scharf, M. and A. Ford, “MPTCP Application Interface Considerations,” October 2009.).
The ability to make effective choices at the sender requires full knowledge of the path "cost", which is unlikely to be the case. There is no mechanism in MPTCP for a receiver to signal their own particular preferences for paths, but this is a necessary feature since receivers will often be the multihomed party, and may have to pay for metered incoming bandwidth. Instead of incorporating complex signalling, it is proposed to use existing TCP features to signal priority implicitly. If a receiver wishes to keep a path active as a backup but wishes to prevent data being sent on that path, it could stop sending ACKs for any data it receives on that path. The sender would interpret this as severe congestion or a broken path and stop using it. We do not advocate this method, however, since this will result in unnecessary retransmissions.
Therefore, a proposal is to use ECN [7] (Ramakrishnan, K., Floyd, S., and D. Black, “The Addition of Explicit Congestion Notification (ECN) to IP,” September 2001.) to to provide fake congestion signals on paths that a receiver wishes to stop being used for data. This has the benefit of causing the sender to back off without the need to retransmit data unnecessarily, as in the case of a lost ACK. This should be sufficient to allow a receiver to express their policy,

although does not permit a rapid increase in throughput when switching to such a path.
TBD: This is clearly an overload of the ECN signal, and as such other solutions, such as explicitly signalling path operation preferences (such as in the reserved bits of certain TCP options, or through entirely new options) may be a preferred solution.

4.5. Closing a Connection

TOC

Under single path TCP, a FIN signifies that the sender has no more data to send. In order to allow subflows to operate independently, however, and with as little change from regular TCP as possible, a FIN in MPTCP only affects the subflow on which it is sent. This allows nodes to exercise considerable freedom over which paths are in use at any one time. The semantics of a FIN remain as for regular TCP, i.e. it is not until both sides have ACKed each other's FINs that the subflow is fully closed.
When an application calls close() on a socket, this indicates that it has no more data to send, and for regular TCP this would result in a FIN on the connection. For MPTCP, an equivalent mechanism is needed, and this is the DATA_FIN. This option, shown in Figure 10 (DATA_FIN option), is attached to a regular FIN option on a subflow.
A DATA_FIN is an indication that the sender has no more data to send, and as such can be used as a rapid indication of the end of data from a sender. A DATA_FIN, as with the FIN on a regular TCP connection, is a unidirectional signal.
A DATA_FIN occupies one octet (the final octet) of Data Sequence Number space. This number is included in the option, and will be ACKed at data level to ensure reliable delivery.
The DATA_FIN is an optimisation to rapidly indicate the end of a data stream and clean up state associated with a MPTCP connection, especially when some subflows may have failed. Specifically, when a DATA_FIN has been received, IF all data has been successfully received, timeouts on all subflows MAY be reduced. Similarly, when sending a DATA_FIN, once all data (including the DATA_FIN, since it occupies one octet of data sequence space) has been acknowledged, FINs must be sent on every subflow. This applies to both endpoints, and is required in order to clean up state in middleboxes.
There are complex interactions, however, between a DATA_FIN and subflow properties:

- A DATA_FIN MUST only be sent on a packet which also has the FIN flag set.

- When DATA_FIN is sent, it should be sent on all subflows.

*There is a one-to-one mapping between the DATA_FIN and the subflow's FIN flag (and its associated sequence space and thus its acknowlegement). In other words, when a subflow's FIN flag has been acknowledged, the associated DATA_FIN is also acknowledged.

*The DATA_ACK (Section 4.4.2 (Data Acknowledgements)), which will be included with a DATA_FIN, is used to verify that all data has been successfully received.

It should be noted that an endpoint may also send a FIN on an individual subflow to shut it down, but this impact is limited to the subflow in question. If all subflows have been closed with a FIN, that is equivalent to having closed the connection with a DATA_FIN.
The full eight-byte data sequence number is always included in a DATA_FIN.

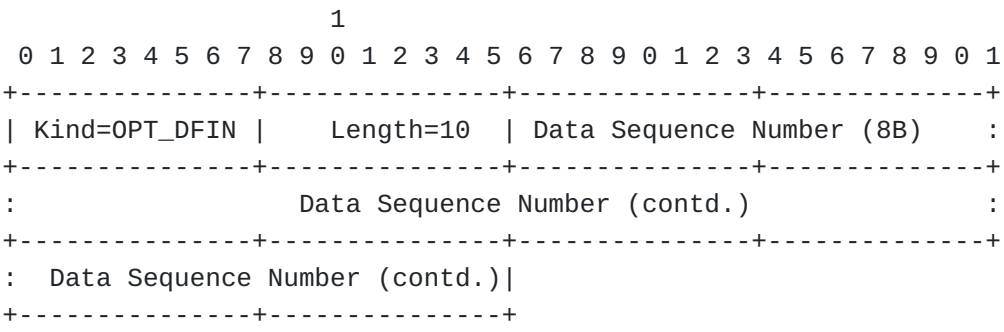

Figure 10: DATA_FIN option

4.6. Error Handling

TOC

TBD
Unknown token in MPTCP SYN should equate to an unknown port, e.g. a TCP reset? We should make this as silent and tolerant as possible. Where possible, we should keep this close to the semantics of TCP. However, some MPTCP-specific issues such as where a data sequence number is missing from a subflow, will definitely need MPTCP-specific errors handling in those cases.

5. Security Considerations

TOC

TBD
(Token generation, handshake mechanisms, new subflow authentication, etc...)
A generic threat analysis for the addition of multipath capabilities to TCP is presented in [8] (Bagnulo, M., “Threat Analysis for Multi-addressed/Multi-path TCP,” February 2010.). The protocol presented here has been designed to minimise or eliminate these identified threats. (A future version of this document will explicitly address the presented threats).
The development of a TCP extension such as this will bring with it many additional security concerns. We have set out here to produce a solution that is "no worse" than current TCP, with the possibility that more secure extensions could be proposed later.
The primary area of concern will be around the handshake to start new subflows which join existing connections. The proposal set out in Section 4.1 (Connection Initiation) and Section 4.2 (Starting a New Subflow) is for the initiator of the new subflow to include the token of the other endpoint in the handshake. The purpose of this is to indicate that the sender of this token was the same entity that received this token at the initial handshake.
One area of concern is that the token could be simply brute-forced. The token must be hard to guess, and as such could be randomly generated. This may still not be strong enough, however, and so the use of 64 bits for the token would alleviate this somewhat.
The two tokens don't need to be the same length. Token B could be 64 bits and token A 32 bits. If JOIN always contains Token B, this would provide adequate security while saving scarce space in the initial SYN, where it is most at a premium.
Use of these tokens only provide an indication that the token is the same as at the initial handshake, and does not say anything about the current sender of the token. Therefore, another approach would be to bring a new measure of freshness in to the handshake, so instead of using the initial token a sender could request a new token from the receiver to use in the next handshake. Hash chains could also be used for this purpose.
Yet another alternative would be for all SYN packets to include a data sequence number. This could either be used as a passive identifier to indicate an awareness of the current data sequence number (although a reasonable window would have to be allowed for delays). Or, the SYN could form part of the data sequence space - but this would cause issues in the event of lost SYNs (if a new subflow is never established), thus causing unnecessary delays for retransmissions.

TOC

6. Interactions with Middleboxes

Multipath TCP will be deployed in a network that no longer provides just basic datagram delivery. A miriad of middleboxes are deployed to optimize various perceived problems with the Internet protocols: NATs primarily address space shortage [9] (Srisuresh, P. and K. Egevang, "Traditional IP Network Address Translator (Traditional NAT)," January 2001.), Performance Enhancing Proxies (PEPs) optimize TCP for different link characteristics [10] (Border, J., Kojo, M., Griner, J., Montenegro, G., and Z. Shelby, "Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations," June 2001.), firewalls [11] (Freed, N., "Behavior of and Requirements for Internet Firewalls," October 2000.) and intrusion detection systems try to block malicious content from reaching a host, and traffic normalizers [12] (Handley, M., Paxson, V., and C. Kreibich, "Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics," 2001.) ensure a consistent view of the traffic stream to IDSes and hosts.

All these middleboxes optimize current applications at the expense of future applications. In effect, future applications must mimic existing ones if they want to be deployed. Further, the precise behaviour of all these middleboxes is not clearly specified, and implementation errors make matters worse, raising the bar for the deployment of new technologies.

Multipath TCP was designed to be deployable in the present world. Its design takes into account "reasonable" existing middlebox behaviour. In this section we outline a few representative middlebox-related failure scenarios and show how multipath TCP handles them. Next, we list the design decisions multipath has made to accomodate the different middleboxes.

A primary concern is our use of new TCP options. Most middleboxes should just forward packets with new options unchanged, yet there are some that don't. These we expect will either strip options and pass the data, drop packets with new options, copy the same option into multiple segments (e.g. when doing segmentation) or drop options during segment coalescing.

MPTCP SYN packets contain the MPC option to indicate the use of MPTCP. When the middlebox drops the packet containing the MPC option either on the outgoing or the return path, the connection will fail. Host A SHOULD fall back to TCP in such cases (studies suggest that few middleboxes drop packets with unknown options). The same applies for subflow setup.

The second case is when the middleboxes strip options. Let's first discuss behaviour for initial connection SYNs (see Figure 11 (Connection Setup with Middleboxes that Strip Options from Packets)). If the option is stripped from the packet on the outgoing path, the connection falls back to regular TCP. If the option is stripped on the return path, host B will wait for a DATA_ACK of its connection SYN, retransmitting the SYN/ACK until it declares the connection failed. Host A thinks it is talking to a regular host, and may send data

segments, but these will not be acked by host B as they do not have the proper mapping. Hence the connection fails. Host A SHOULD fall back to regular TCP after the connection times out.
Subflow SYNs contain the OPT_JOIN option. If this option is stripped on the outgoing path the SYN will appear to be a regular SYN to host B. Depending on whether there is a listening socket on the target port, host B will reply either with SYN/ACK or RST (subflow connection fails). When host A receives the SYN/ACK it sends a RST because the SYN/ACK does not contain the OPT_JOIN option and its token. Either way, the connection fails.

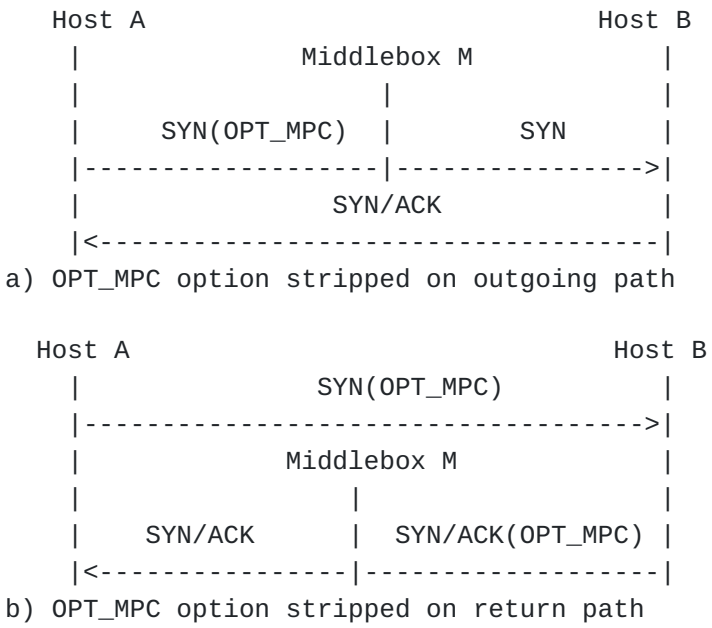

Figure 11: Connection Setup with Middleboxes that Strip Options from Packets

---

We now examine data flow with MPTCP, assuming the flow is correctly setup which implies the options in the SYN packets were allowed through by the relevant middleboxes. If options are allowed through and there is no resegmentation or coalescing to TCP segments, multipath TCP flows can proceed without problems.
If options are stripped in either direction by middleboxes (this is unlikely, as the SYN options did get through), the particular subflow will timeout repeatedly while waiting for a DATA_ACK or subflow-level ACK, and will be closed. If the subflow is the initial one, host A SHOULD fall back to regular TCP.
We can further analyze what happens when a fraction of options is stripped. The multipath subflow should survive losing a fraction of DATA_ACKs and data sequence mappings, but performance will degrade as the fraction of stripped options increases. We do not expect such cases

to appear in practice, though: most middleboxes will either strip all options or let them all through.
We end this section with a list of middlebox classes, their behaviour and the elements in the MPTCP design that allow operation through such middleboxes. Issues surrounding dropping packets with options or stripping options were discussed above, and are not included here:

*NAT: will prevent flow/subflow setup when the server does not have a public address. MPTCP assumes the server has at least one public address (or the client uses standard NAT traversal to reach it) that is used to setup the connection. If uses Add Address messages to signal the existence of other addresses.

*Performance Enhancing Proxies: might pro-actively ACK data and then fail. MPTCP uses the DATA_ACK to make progress when one of its subflows fails in this way. This is why MPTCP does not use subflow ACKs to infer connection level ACKs.

*Traffic Normalizers: do not allow holes in sequence numbers, cache packets and retransmit the same data. MPTCP looks like standard TCP on the wire, and will not retransmit different data on the same subflow sequence number.

*Segmentation/Coalescing (e.g. tcp segmentation offloading, etc): might copy options between packets and might strip some options. MPTCP's data sequence mapping includes the subflow sequence number instead of using the sequence number in the segment. In this way, the mapping is independent of the packets that carry it.

*Firewalls: might perform sequence number randomization on outgoing connections. MPTCP uses relative sequence numbers in data sequence mapping to cope with this.

---

7. Interfaces

TOC

TBD
Interface with applications, interface with TCP, interface with lower layers...
Discussion of interaction with applications (both in terms of how MPTCP will affect an application's assumptions of the transport layer, and what API extensions an application may wish to use with MPTCP) are discussed in [5] (Scharf, M. and A. Ford, “MPTCP Application Interface Considerations,” October 2009.).

---

8. Open Issues

TOC

This specification is a work-in-progress, and as such there are many issues that are still to be resolved. This section lists many of the key open issues within this specification; these are discussed in more detail in the appropriate sections throughout this document.

- Best handshake mechanisms (Section 4.1 (Connection Initiation)). This document contains a proposed scheme by which connections and subflows can be set up. It is felt that, although this is "no worse than regular TCP", there could be opportunities for significant improvements in security that could be included (potentially optionally) within this protocol.

- Issues around simultaneous opens, where both ends attempt to create a new subflow simultaneously, need to be investigated and behaviour specified.

- Appropriate mechanisms for controlling policy/priority of subflow usage (specifically regarding controlling incoming traffic, Section 4.4.6 (Subflow Policy)). The ECN signal is currently proposed but other alternatives, including per subflow receive windows or options indicating path properties, could be employed instead.

- How much control do we want over subflows from other subflows (e.g. closing when interface has failed)? Do we want to differentiate between subflows and addresses (Section 4.2 (Starting a New Subflow))?

- Do we want a connection identifier in every packet? E.g. would it make the implementation of an IDS easier?

- Should we do signaling in the TCP payload, rather than options as proposed in this draft? We discuss this alternative in the appendix.

- Should we explicitly support SYN cookies? With the current design, MPTCP would be downgraded to basic TCP if SYN cookies were used. Is it worth designing the protocol to allow stateless server handshake?

- Instead of an Address ID in JOIN, are there any cases where a Subflow ID (i.e. unique to the subflow) would be useful instead? For example, two addresses which become NATted to the same address?

9. Acknowledgements

TOC

The authors are supported by Trilogy (http://www.trilogy-project.org), a research project (ICT-216372) partially funded by the European Community under its Seventh Framework Program. The views expressed here are those of the author(s) only. The European Commission is not liable for any use that may be made of the information in this document.
The authors gratefully acknowledge significant input into this document from many members of the Trilogy project, notably Iljitsch van Beijnum, Lars Eggert, Marcelo Bagnulo Braun, Robert Hancock, Pasi Sarolahti, Olivier Bonaventure, Toby Moncaster, Philip Eardley, Andrew McDonald and Sergio Lembo.

---

10. IANA Considerations

TOC

This document will make a request to IANA to allocate new values for TCP Option identifiers, as follows:

Symbol	Name	Ref	Value
OPT_MPCAP	Multipath Capable	Section 4.1 (Connection Initiation)	(tbc)
OPT_ADDR	Add Address	Section 4.3.1 (Address Advertisement)	(tbc)
OPT_REMADR	Remove Address	Section 4.3.2 (Remove Address)	(tbc)
OPT_JOIN	Join Connection	Section 4.2 (Starting a New Subflow)	(tbc)
OPT_DSN	Data Sequence Mapping	Section 4.4 (General MPTCP Operation)	(tbc)
OPT_DACK	DATA_ACK	Section 4.4 (General MPTCP Operation)	(tbc)
OPT_DFIN	DATA_FIN	Section 4.5 (Closing a Connection)	(tbc)

Table 1: TCP Options for MPTCP

TOC

11. References

11.1. Normative References

TOC

[1]	Bradner, S., “Key words for use in RFCs to Indicate Requirement Levels,” BCP 14, RFC 2119, March 1997 (TXT, HTML, XML).

11.2. Informative References

TOC

[2]	Postel, J., “Transmission Control Protocol,” STD 7, RFC 793, September 1981 (TXT).
[3]	Ford, A., Raiciu, C., Barre, S., and J. Iyengar, “Architectural Guidelines for Multipath TCP Development,” draft-ietf-mptcp-architecture-00 (work in progress), March 2010 (TXT).
[4]	Raiciu, C., Handley, M., and D. Wischik, “Coupled Multipath-Aware Congestion Control,” draft-raiciu-mptcp-congestion-00 (work in progress), October 2009 (TXT).
[5]	Scharf, M. and A. Ford, “MPTCP Application Interface Considerations,” draft-scharf-mptcp-api-00 (work in progress), October 2009 (TXT).
[6]	Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, “TCP Selective Acknowledgment Options,” RFC 2018, October 1996 (TXT, HTML, XML).
[7]	Ramakrishnan, K., Floyd, S., and D. Black, “The Addition of Explicit Congestion Notification (ECN) to IP,” RFC 3168, September 2001 (TXT).
[8]	Bagnulo, M., “Threat Analysis for Multi-addressed/Multi-path TCP,” draft-ietf-mptcp-threat-00 (work in progress), February 2010 (TXT).
[9]	Srisuresh, P. and K. Egevang, “Traditional IP Network Address Translator (Traditional NAT),” RFC 3022, January 2001 (TXT).
[10]	Border, J., Kojo, M., Griner, J., Montenegro, G., and Z. Shelby, “Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations,” RFC 3135, June 2001 (TXT).
[11]	Freed, N., “Behavior of and Requirements for Internet Firewalls,” RFC 2979, October 2000 (TXT).
[12]	Handley, M., Paxson, V., and C. Kreibich, “Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics,” Usenix Security 2001, 2001.
[13]	Eddy, W. and A. Langley, “Extending the Space Available for TCP Options,” draft-eddy-tcp-loo-04 (work in progress), July 2008 (TXT).

Appendix A. Notes on use of TCP Options

TOC

The TCP option space is limited due to the length of the Data Offset field in the TCP header (4 bits), which defines the TCP header length in 32-bit words. With the standard TCP header being 20 bytes, this leaves a maximum of 40 bytes for options, and many of these may already be used by options such as timestamp and SACK.

We have performed a brief study on the commonly used TCP options in both SYN, data packets and pure ACK packets, and found that there is enough room to fit all the options we propose using in this draft.

SYN packets typically include MSS, window scale, sack permitted and timestamp options. Together these sum to 19B. The multipath capable (MPC) option requires a max of 16B, and the Join option requires 8B, so they both fit the existing space.

TCP data packets typically carry timestamp options in every packet, taking 10B. That leaves 30B which are enough to encode the data sequence mapping (max 16B) and the DATA_ACK if the flow is bidirectional (max 10B).

Pure ACKs in TCP typically contain only timestamps (10B). Here, multipath TCP typically needs to encode the DATA_ACK (max 10B). Ocasionally acks will contain SACK information. Depending on the number of lost packets, SACK may utilize the entire option space. We propose reducing the number of SACK blocks by one to accomodate the DATA_ACK.

Encoding Add/Remove address options uses at most 10B (for IPv6 addresses). These will fit in data packets if the DATA_ACK is not present. Otherwise, the endpoint can insert pure ACKs that contain the add address option. Finally, if SACK information is included in the data packets, one further block can be removed to accomodate the add address option.

All the new options fit in the space available yet there is little room for adding new options. We note that if 8B data sequence numbers are used, PAWS is no longer needed. Hence the use for timestamps is limited to providing RTT measurements for retransmitted packets. As loss rates are typically low, we believe we can just stop using timestamps, claiming 10B of options space on all packets.

Alternatively, we could use a TCP option to increase the option space, such as that proposed in [13] (Eddy, W. and A. Langley, “Extending the Space Available for TCP Options,” July 2008.). The proposal extends the 4 bit header to 16 bits. Such an option could only be used between nodes that support it, however, and so long options could not be used until a handshake is complete.

Finally, there are issues with options reliability. As options can also be sent on pure ACKs, these are not reliably sent. This is not an issue for DATA_ACK due to their cumulative nature, but may be an issue for add/remove address options. Here we favour redundant transmissions at the sender (whether on multiple paths, or on the same path on a number

of ACKs). The cases where options are stripped by middleboxes are discussed in Section 6 (Interactions with Middleboxes).

Appendix B. Signaling Control Information in the Payload

TOC

Appendix C. Resync Packet

TOC

In earlier versions of this draft, we proposed the use of a "re-sync" option that would be used in certain circumstances when a sender needs to instruct the receiver to skip over certain subflow sequence numbers (i.e. to treat the specified sequence space as having been received and acknowledged).

The typical use of this option will be when packets are retransmitted on different subflows, after failing to be acknowledged on the original subflow. In such a case, it becomes necessary to move forward the original subflow's sequence numbering so as not to later transmit different data with a previously used sequence number (i.e. when more data comes to be transmitted on the original subflow, it would be different data, and so must not be sent with previously-used (but unacknowledged) sequence numbering).

The rationale for needing to do this is two-fold: firstly, when ACKs are received they are for the subflow only, and the sender infers from this the data that was sent - if the same sequence space could be occupied by different data, the sender won't know whether the intended data was received. Secondly, certain classes of middleboxes may cache data and not send the new data on a previously-seen sequence number. This option was dropped, however, since some middleboxes may get confused when they meet a hole in the sequence space, and do not understand the resync option. It is therefore felt that the same data must continue to be retransmitted on a subflow even if it is already received after being retransmitted on another. There should not be a significant performance hit from this since the amount of data involved and needing to be retransmitted multiple times will be relatively small.

Therefore, it is necessary to 're-sync' the expected sequence numbering at the receiving end of a subflow, using the following TCP option. This packet declares a sequence number space (inclusive) which the receiving node should skip over, i.e. if the receiver's next expected sequence number was previously within the range start_seq_num to end_seq_num, move it forward to end_seq_num + 1.

This option will be used on the first new packet on the subflow that needs its sequence numbering re-synchronised. It will be continue to be included on every packet sent on this subflow until a packet containing

this option has been acknowledged (i.e. if subflow acknowledgements exist for packets beyond the end sequence number). If the end sequence number is earlier than the current expected sequence number (i.e. if a resync packet has already been received), this option should be ignored.

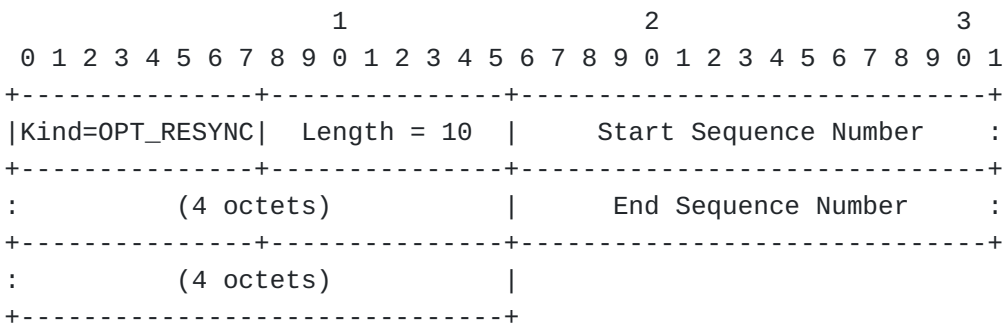

Figure 12: Resync option

Appendix D. Changelog

TOC

This section maintains logs of significant changes made to this document between versions.

D.1. Changes since draft-ford-mptcp-multiaddressed-02

TOC

*Remote Version and Address ID in MP_CAPABLE in Section 4.1 (Connection Initiation), and make ISN be 6 bytes.

*Data sequence numbers are now always 8 bytes. But in some cases where it is unambiguous it is permissible to only send the lower 4 bytes if space is at a premium.

*Clarified behaviour of OPT_JOIN in Section 4.2 (Starting a New Subflow).

*Added DATA_ACK to Section 4.4 (General MPTCP Operation).

*Clarified fallback to non-multipath once a non-MP-capable SYN is sent.

Authors' Addresses

TOC

	Alan Ford
	Roke Manor Research
	Old Salisbury Lane
	Romsey, Hampshire SO51 0ZN
	UK
Phone:	+44 1794 833 465
Email:	alan.ford@roke.co.uk
	Costin Raiciu
	University College London
	Gower Street
	London WC1E 6BT
	UK
Email:	c.raiciu@cs.ucl.ac.uk
	Mark Handley
	University College London
	Gower Street
	London WC1E 6BT
	UK
Email:	m.handley@cs.ucl.ac.uk