          TCP Extensions for Multipath Operation with Multiple Addresses

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 (i.e., reliable
   bytestream), and it provides the components necessary to establish
   and use multiple TCP flows across potentially disjoint paths.

Table of Contents

1. Introduction

   Multipath TCP (MPTCP) is a set of extensions to regular TCP [1] to
   provide a Multipath TCP [2] service, which enables a transport
   connection to operate across multiple paths simultaneously. This
   document presents the protocol changes required to add multipath
   capability to TCP; specifically, those for signaling 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 three others:

   o Architecture [2], which explains the motivations behind Multipath
      TCP, contains a discussion of high-level design decisions on which
      this design is based, and an explanation of a functional
      separation through which an extensible MPTCP implementation can be
      developed.

   o Congestion control [5] presents a safe congestion control
      algorithm for coupling the behavior of the multiple paths in order
      to "do no harm" to other network users.

   o Application considerations [6] discusses 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.

1.1. Design Assumptions

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

   o It must be backwards-compatible with current, regular TCP, to
      increase its chances of deployment.

   o It can be assumed that one or both hosts are multihomed and
      multiaddressed.

   To simplify the design, we assume that the presence of multiple
   addresses at a host 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
   utilization and resilience to a subset of node failures. The
   congestion control algorithms defined in [5] ensure this does not act
   detrimentally. Furthermore, there may be some scenarios where
   different TCP ports on a single host can provide disjoint paths (such

   as through certain Equal-Cost Multipath (ECMP) implementations [7]),
   and so the MPTCP design also supports the use of ports in path
   identifiers.

   There are three aspects to the backwards-compatibility listed above
   (discussed in more detail in [2]):

   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; TCP options may be
      removed or duplicated.

   Application Constraints: The protocol must be usable with no change
      to existing applications that use the common TCP API (although it
      is reasonable that not all features would be available to such
      legacy applications). Furthermore, the protocol must provide the
      same service model as regular TCP to the application.

   Fallback: 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.

   The complementary application considerations document [6] discusses
   the necessary features of an API to provide backwards-compatibility,
   as well as API extensions to convey the behavior of MPTCP at a level
   of control and information equivalent to that available with regular,
   single-path TCP.

   Further discussion of the design constraints and associated design
   decisions are given in the MPTCP Architecture document [2] and in
   [8].

1.2. Multipath TCP in the Networking Stack

   MPTCP operates at the transport layer and aims to be transparent to
   both higher and lower layers. It is a set of additional features on
   top of standard TCP; Figure 1 illustrates this layering. MPTCP is
   designed to be usable by legacy applications with no changes;
   detailed discussion of its interactions with applications is given in
   [6].

```
                             +------------------------------+
                             |          Application         |
           +---------------+ +------------------------------+
           |  Application  | |             MPTCP            |
           +---------------+ +- - - - - - - +- - - - - - - -+
           |      TCP      | | Subflow (TCP) | Subflow (TCP) |
           +---------------+ +------------------------------+
           |      IP       | |      IP       |      IP       |
           +---------------+ +------------------------------+
```

        Figure 1: Comparison of Standard TCP and MPTCP Protocol Stacks

1.3. Terminology

   This document makes use of a number of terms that are either MPTCP-
   specific or have defined meaning in the context of MPTCP, as follows:

   Path: A sequence of links between a sender and a receiver, defined
      in this context by a 4-tuple of source and destination address/
      port pairs.

   Subflow: A flow of TCP segments operating over an individual path,
      which forms part of a larger MPTCP connection. A subflow is
      started and terminated similar to a regular TCP connection.

   (MPTCP) Connection: A set of one or more subflows, over which an
      application can communicate between two hosts. There is a one-to-
      one mapping between a connection and an application socket.

   Data-level: The payload data is nominally transferred 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 a host. May also be referred to as a "Connection ID".

   Host: An end host operating an MPTCP implementation, and either
      initiating or accepting an MPTCP connection.

   In addition to these terms, note that MPTCP's interpretation of, and
   effect on, regular single-path TCP semantics are discussed in
   Section 4.

1.4. MPTCP Concept

   This section provides a high-level summary of normal operation of
   MPTCP, and is illustrated by the scenario shown in Figure 2. A
   detailed description of operation is given in Section 3.

   o To a non-MPTCP-aware application, MPTCP will behave the same as
      normal TCP. Extended APIs could provide additional control to
      MPTCP-aware applications [6]. An application begins by opening a
      TCP socket in the normal way. MPTCP signaling and operation are
      handled by the MPTCP implementation.

   o An MPTCP connection begins similarly to a regular TCP connection.
      This is illustrated in Figure 2 where an MPTCP connection is
      established between addresses A1 and B1 on Hosts A and B,
      respectively.

   o If extra paths are available, additional TCP sessions (termed
      MPTCP "subflows") 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.

   o MPTCP identifies multiple paths by the presence of multiple
      addresses at hosts. 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.

   o The discovery and setup of additional subflows will be achieved
      through a path management method; this document describes a
      mechanism by which a host can initiate new subflows by using its
      own additional addresses, or by signaling its available addresses
      to the other host.

   o MPTCP adds connection-level sequence numbers to allow the
      reassembly of segments arriving on multiple subflows with
      differing network delays.

   o Subflows are terminated as regular TCP connections, with a four-
      way FIN handshake. The MPTCP connection is terminated by a
      connection-level FIN.

```
              Host A                            Host B
        ------------------------        ------------------------
        Address A1     Address A2       Address B1     Address B2
        ----------     ----------       ----------     ----------
           |              |                |              |
           |       (initial connection setup)            |
           |------------------------------------>|       |
           |<------------------------------------|       |
           |              |                |              |
           |             (additional subflow setup)      |
           |              |-------------------->|         |
           |              |<--------------------|         |
           |              |                |              |
           |              |                |              |
           |              |                |              |
```
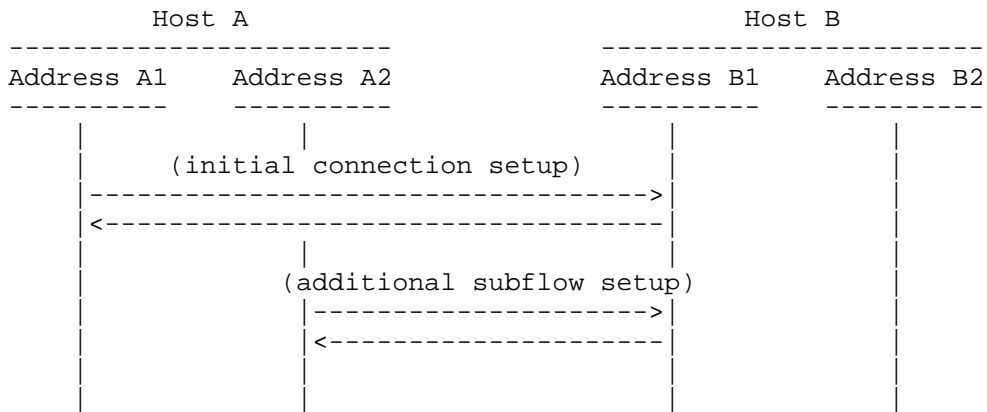
                 Figure 2: Example MPTCP Usage Scenario

1.5. Requirements Language

   The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
   "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
   document are to be interpreted as described in RFC 2119 [3].

2. Operation Overview

   This section presents a single description of common MPTCP operation,
   with reference to the protocol operation. This is a high-level
   overview of the key functions; the full specification follows in
   Section 3. Extensibility and negotiated features are not discussed
   here. Considerable reference is made to symbolic names of MPTCP
   options throughout this section -- these are subtypes of the IANA-
   assigned MPTCP option (see Section 8), and their formats are defined
   in the detailed protocol specification that follows in Section 3.

   A Multipath TCP connection provides a bidirectional bytestream
   between two hosts communicating like normal TCP and, thus, does not
   require any change to the applications. However, Multipath TCP
   enables the hosts to use different paths with different IP addresses
   to exchange packets belonging to the MPTCP connection. A Multipath
   TCP connection appears like a normal TCP connection to an
   application. However, to the network layer, each MPTCP subflow looks
   like a regular TCP flow whose segments carry a new TCP option type.
   Multipath TCP manages the creation, removal, and utilization of these
   subflows to send data. The number of subflows that are managed
   within a Multipath TCP connection is not fixed and it can fluctuate
   during the lifetime of the Multipath TCP connection.

All MPTCP operations are signaled with a TCP option -- a single
numerical type for MPTCP, with "sub-types" for each MPTCP message.
What follows is a summary of the purpose and rationale of these
messages.

2.1. Initiating an MPTCP Connection

This is the same signaling as for initiating a normal TCP connection,
but the SYN, SYN/ACK, and ACK packets also carry the MP_CAPABLE
option. This is variable length and serves multiple purposes.
Firstly, it verifies whether the remote host supports Multipath TCP;
secondly, this option allows the hosts to exchange some information
to authenticate the establishment of additional subflows. Further
details are given in Section 3.1.

```
   Host A                                Host B
   ------                                ------
   MP_CAPABLE              ->
   [A's key, flags]
                          <-             MP_CAPABLE
                                         [B's key, flags]
   ACK + MP_CAPABLE       ->
   [A's key, B's key, flags]
```

2.2. Associating a New Subflow with an Existing MPTCP Connection

The exchange of keys in the MP_CAPABLE handshake provides material
that can be used to authenticate the endpoints when new subflows will
be set up. Additional subflows begin in the same way as initiating a
normal TCP connection, but the SYN, SYN/ACK, and ACK packets also
carry the MP_JOIN option.

Host A initiates a new subflow between one of its addresses and one
of Host B's addresses. The token -- generated from the key -- is
used to identify which MPTCP connection it is joining, and the HMAC
is used for authentication. The Hash-based Message Authentication
Code (HMAC) uses the keys exchanged in the MP_CAPABLE handshake, and
the random numbers (nonces) exchanged in these MP_JOIN options.
MP_JOIN also contains flags and an Address ID that can be used to
refer to the source address without the sender needing to know if it
has been changed by a NAT. Further details are in Section 3.2.

```
    Host A                                      Host B
    ------                                      ------
    MP_JOIN               ->
    [B's token, A's nonce,
     A's Address ID, flags]
                          <-             MP_JOIN
                                         [B's HMAC, B's nonce,
                                          B's Address ID, flags]
    ACK + MP_JOIN         ->
    [A's HMAC]

                          <-             ACK
```

2.3. Informing the Other Host about Another Potential Address

   The set of IP addresses associated to a multihomed host may change
   during the lifetime of an MPTCP connection. MPTCP supports the
   addition and removal of addresses on a host both implicitly and
   explicitly. If Host A has established a subflow starting at address
   IP#-A1 and wants to open a second subflow starting at address IP#-A2,
   it simply initiates the establishment of the subflow as explained
   above. The remote host will then be implicitly informed about the
   new address.

   In some circumstances, a host may want to advertise to the remote
   host the availability of an address without establishing a new
   subflow, for example, when a NAT prevents setup in one direction. In
   the example below, Host A informs Host B about its alternative IP
   address (IP#-A2). Host B may later send an MP_JOIN to this new
   address. Due to the presence of middleboxes that may translate IP
   addresses, this option uses an address identifier to unambiguously
   identify an address on a host. Further details are in Section 3.4.1.

```
    Host A                                      Host B
    ------                                      ------
    ADD_ADDR                 ->
    [IP#-A2,
     IP#-A2's Address ID]
```

   There is a corresponding signal for address removal, making use of
   the Address ID that is signaled in the add address handshake.
   Further details in Section 3.4.2.

```
    Host A                                      Host B
    ------                                      ------
    REMOVE_ADDR              ->
    [IP#-A2's Address ID]
```

2.4. Data Transfer Using MPTCP

   To ensure reliable, in-order delivery of data over subflows that may
   appear and disappear at any time, MPTCP uses a 64-bit data sequence
   number (DSN) to number all data sent over the MPTCP connection. Each
   subflow has its own 32-bit sequence number space and an MPTCP option
   maps the subflow sequence space to the data sequence space. In this
   way, data can be retransmitted on different subflows (mapped to the
   same DSN) in the event of failure.

   The "Data Sequence Signal" carries the "Data Sequence Mapping". The
   data sequence mapping consists of the subflow sequence number, data
   sequence number, and length for which this mapping is valid. This
   option can also carry a connection-level acknowledgment (the "Data
   ACK") for the received DSN.

   With MPTCP, all subflows share the same receive buffer and advertise
   the same receive window. There are two levels of acknowledgment in
   MPTCP. Regular TCP acknowledgments are used on each subflow to
   acknowledge the reception of the segments sent over the subflow
   independently of their DSN. In addition, there are connection-level
   acknowledgments for the data sequence space. These acknowledgments
   track the advancement of the bytestream and slide the receiving
   window.

   Further details are in Section 3.3.

      Host A   Host B
      ------   ------
      DATA_SEQUENCE_SIGNAL   ->
      [Data Sequence Mapping]
      [Data ACK]
      [Checksum]

2.5. Requesting a Change in a Path's Priority

   Hosts can indicate at initial subflow setup whether they wish the
   subflow to be used as a regular or backup path -- a backup path only
   being used if there are no regular paths available. During a
   connection, Host A can request a change in the priority of a subflow
   through the MP_PRIO signal to Host B. Further details are in
   Section 3.3.8.

      Host A   Host B
      ------   ------
      MP_PRIO   ->

2.6. Closing an MPTCP Connection

   When Host A wants to inform Host B that it has no more data to send,
   it signals this "Data FIN" as part of the Data Sequence Signal (see
   above). It has the same semantics and behavior as a regular TCP FIN,
   but at the connection level. Once all the data on the MPTCP
   connection has been successfully received, then this message is
   acknowledged at the connection level with a DATA_ACK. Further
   details are in Section 3.3.3.

        Host A   Host B
        ------   ------
        DATA_SEQUENCE_SIGNAL   ->
        [Data FIN]

                                  <-   (MPTCP DATA_ACK)

2.7. Notable Features

   It is worth highlighting that MPTCP's signaling has been designed
   with several key requirements in mind:

   o To cope with NATs on the path, addresses are referred to by
      Address IDs, in case the IP packet's source address gets changed
      by a NAT. Setting up a new TCP flow is not possible if the
      passive opener is behind a NAT; to allow subflows to be created
      when either end is behind a NAT, MPTCP uses the ADD_ADDR message.

   o MPTCP falls back to ordinary TCP if MPTCP operation is not
      possible, for example, if one host is not MPTCP capable or if a
      middlebox alters the payload.

   o To meet the threats identified in [9], the following steps are
      taken: keys are sent in the clear in the MP_CAPABLE messages;
      MP_JOIN messages are secured with HMAC-SHA1 ([10], [4]) using
      those keys; and standard TCP validity checks are made on the other
      messages (ensuring sequence numbers are in-window).

3. MPTCP Protocol

   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 signaled using optional TCP header fields.
   A single TCP option number ("Kind") has been assigned by IANA for
   MPTCP (see Section 8), and then individual messages will be
   determined by a "subtype", the values of which are also stored in an
   IANA registry (and are also listed in Section 8).

Throughout this document, when reference is made to an MPTCP option
by symbolic name, such as "MP_CAPABLE", this refers to a TCP option
with the single MPTCP option type, and with the subtype value of the
symbolic name as defined in Section 8. This subtype is a 4-bit field
-- the first 4 bits of the option payload, as shown in Figure 3. The
MPTCP messages are defined in the following sections.

```
                          1                   2                   3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
     +---------------+---------------+-------+----------------------+
     |     Kind      |    Length     |Subtype|                      |
     +---------------+---------------+-------+                      |
     |                     Subtype-specific data                   |
     |                       (variable length)                     |
     +-------------------------------------------------------------+
```
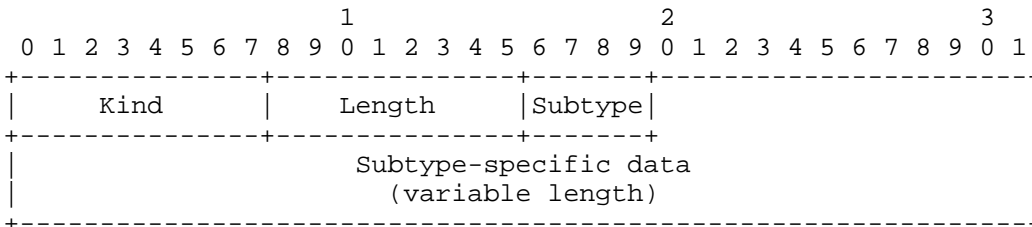
Figure 3: MPTCP Option Format

Those MPTCP options associated with subflow initiation are used on
packets with the SYN flag set. Additionally, there is one MPTCP
option for signaling metadata to ensure segmented data can be
recombined for delivery to the application.

The remaining options, however, are signals that do not need to be on
a specific packet, such as those for signaling additional addresses.
Whilst an implementation may desire to send MPTCP options as soon as
possible, it may not be possible to combine all desired options (both
those for MPTCP and for regular TCP, such as SACK (selective
acknowledgment) [11]) on a single packet. Therefore, an
implementation may choose to send duplicate ACKs containing the
additional signaling information. This changes the semantics of a
duplicate ACK; these are usually only sent as a signal of a lost
segment [12] in regular TCP. Therefore, an MPTCP implementation
receiving a duplicate ACK that contains an MPTCP option MUST NOT
treat it as a signal of congestion. Additionally, an MPTCP
implementation SHOULD NOT send more than two duplicate ACKs in a row
for the purposes of sending MPTCP options alone, in order to ensure
no middleboxes misinterpret this as a sign of congestion.

Furthermore, standard TCP validity checks (such as ensuring the
sequence number and acknowledgment number are within window) MUST be
undertaken before processing any MPTCP signals, as described in [13].

3.1. Connection Initiation

   Connection initiation begins with a SYN, SYN/ACK, ACK exchange on a
   single path. Each packet contains the Multipath Capable (MP_CAPABLE)
   TCP option (Figure 4). This option declares its sender is capable of
   performing Multipath TCP and wishes to do so on this particular
   connection.

   This option is used to declare the 64-bit key that the sender has
   generated for this MPTCP connection. This key is used to
   authenticate the addition of future subflows to this connection.
   This is the only time the key will be sent in clear on the wire
   (unless "fast close", Section 3.5, is used); all future subflows will
   identify the connection using a 32-bit "token". This token is a
   cryptographic hash of this key. The algorithm for this process is
   dependent on the authentication algorithm selected; the method of
   selection is defined later in this section.

   This key is generated by its sender, and its method of generation is
   implementation specific. The key MUST be hard to guess, and it MUST
   be unique for the sending host at any one time. Recommendations for
   generating random numbers for use in keys are given in [14].
   Connections will be indexed at each host by the token (a one-way hash
   of the key). Therefore, an implementation will require a mapping
   from each token to the corresponding connection, and in turn to the
   keys for the connection.

   There is a risk that two different keys will hash to the same token.
   The risk of hash collisions is usually small, unless the host is
   handling many tens of thousands of connections. Therefore, an
   implementation SHOULD check its list of connection tokens to ensure
   there is not a collision before sending its key in the SYN/ACK. This
   would, however, be costly for a server with thousands of connections.
   The subflow handshake mechanism (Section 3.2) will ensure that new
   subflows only join the correct connection, however, through the
   cryptographic handshake, as well as checking the connection tokens in
   both directions, and ensuring sequence numbers are in-window. So in
   the worst case if there was a token collision, the new subflow would
   not succeed, but the MPTCP connection would continue to provide a
   regular TCP service.

   The MP_CAPABLE option is carried on the SYN, SYN/ACK, and ACK packets
   that start the first subflow of an MPTCP connection. The data
   carried by each packet is as follows, where A = initiator and B =
   listener.

o SYN (A->B): A's Key for this connection.

o SYN/ACK (B->A): B's Key for this connection.

o ACK (A->B): A's Key followed by B's Key.

The contents of the option is determined by the SYN and ACK flags of
the packet, verified by the option's length field. For the diagram
shown in Figure 4, "sender" and "receiver" refer to the sender or
receiver of the TCP packet (which can be either host). If the SYN
flag is set, a single key is included; if only an ACK flag is set,
both keys are present.

B's Key is echoed in the ACK in order to allow the listener (Host B)
to act statelessly until the TCP connection reaches the ESTABLISHED
state. If the listener acts in this way, however, it MUST generate
its key in a way that would allow it to verify that it generated the
key when it is echoed in the ACK.

This exchange allows the safe passage of MPTCP options on SYN packets
to be determined. If any of these options are dropped, MPTCP will
gracefully fall back to regular single-path TCP, as documented in
Section 3.6. Note that new subflows MUST NOT be established (using
the process documented in Section 3.2) until a Digital Signature
Standard (DSS) option has been successfully received across the path
(as documented in Section 3.3).

```
                         1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +---------------+---------------+-------+-------+---------------+
    |     Kind      |    Length     |Subtype|Version|A|B|C|D|E|F|G|H|
    +---------------+---------------+-------+-------+---------------+
    |                   Option Sender's Key (64 bits)               |
    |                                                               |
    |                                                               |
    +---------------------------------------------------------------+
    |                  Option Receiver's Key (64 bits)              |
    |                      (if option Length == 20)                 |
    |                                                               |
    +---------------------------------------------------------------+
```
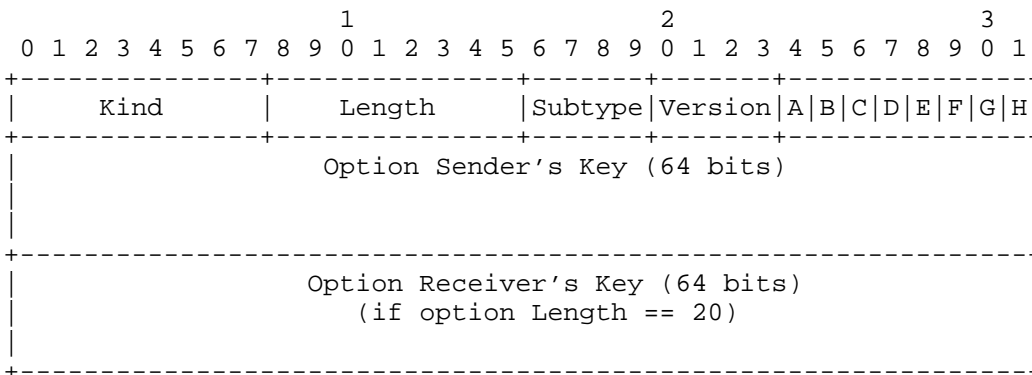
          Figure 4: Multipath Capable (MP_CAPABLE) Option

The first 4 bits of the first octet in the MP_CAPABLE option
(Figure 4) define the MPTCP option subtype (see Section 8; for
MP_CAPABLE, this is 0), and the remaining 4 bits of this octet
specify the MPTCP version in use (for this specification, this is 0).

The second octet is reserved for flags, allocated as follows:

A: The leftmost bit, labeled "A", SHOULD be set to 1 to indicate
    "Checksum Required", unless the system administrator has decided
    that checksums are not required (for example, if the environment
    is controlled and no middleboxes exist that might adjust the
    payload).

B: The second bit, labeled "B", is an extensibility flag, and MUST be
    set to 0 for current implementations. This will be used for an
    extensibility mechanism in a future specification, and the impact
    of this flag will be defined at a later date. If receiving a
    message with the 'B' flag set to 1, and this is not understood,
    then this SYN MUST be silently ignored; the sender is expected to
    retry with a format compatible with this legacy specification.
    Note that the length of the MP_CAPABLE option, and the meanings of
    bits "C" through "H", may be altered by setting B=1.

C through H: The remaining bits, labeled "C" through "H", are used
    for crypto algorithm negotiation. Currently only the rightmost
    bit, labeled "H", is assigned. Bit "H" indicates the use of HMAC-
    SHA1 (as defined in Section 3.2). An implementation that only
    supports this method MUST set bit "H" to 1, and bits "C" through
    "G" to 0.

A crypto algorithm MUST be specified. If flag bits C through H are
all 0, the MP_CAPABLE option MUST be treated as invalid and ignored
(that is, it must be treated as a regular TCP handshake).

The selection of the authentication algorithm also impacts the
algorithm used to generate the token and the initial data sequence
number (IDSN). In this specification, with only the SHA-1 algorithm
(bit "H") specified and selected, the token MUST be a truncated (most
significant 32 bits) SHA-1 hash ([4], [15]) of the key. A different,
64-bit truncation (the least significant 64 bits) of the SHA-1 hash
of the key MUST be used as the initial data sequence number. Note
that the key MUST be hashed in network byte order. Also note that
the "least significant" bits MUST be the rightmost bits of the SHA-1
digest, as per [4]. Future specifications of the use of the crypto
bits may choose to specify different algorithms for token and IDSN
generation.

Both the crypto and checksum bits negotiate capabilities in similar
ways. For the Checksum Required bit (labeled "A"), if either host
requires the use of checksums, checksums MUST be used. In other
words, the only way for checksums not to be used is if both hosts in
their SYNs set A=0. This decision is confirmed by the setting of the
"A" bit in the third packet (the ACK) of the handshake. For example,

if the initiator sets A=0 in the SYN, but the responder sets A=1 in
the SYN/ACK, checksums MUST be used in both directions, and the
initiator will set A=1 in the ACK. The decision whether to use
checksums will be stored by an implementation in a per-connection
binary state variable.

For crypto negotiation, the responder has the choice. The initiator
creates a proposal setting a bit for each algorithm it supports to 1
(in this version of the specification, there is only one proposal, so
bit "H" will be always set to 1). The responder responds with only 1
bit set -- this is the chosen algorithm. The rationale for this
behavior is that the responder will typically be a server with
potentially many thousands of connections, so it may wish to choose
an algorithm with minimal computational complexity, depending on the
load. If a responder does not support (or does not want to support)
any of the initiator's proposals, it can respond without an
MP_CAPABLE option, thus forcing a fallback to regular TCP.

The MP_CAPABLE option is only used in the first subflow of a
connection, in order to identify the connection; all following
subflows will use the "Join" option (see Section 3.2) to join the
existing connection.

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; thus,
the MPTCP session MUST operate as a 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 third packet (the ACK) does not
contain the MP_CAPABLE option, then the session MUST fall back to
operating as a regular, single-path TCP. This is to maintain
compatibility with middleboxes on the path that drop some or all TCP
options. Note that an implementation MAY choose to attempt sending
MPTCP options more than one time before making this decision to
operate as regular TCP (see Section 3.8).

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. It is possible that MPTCP and
non-MPTCP SYNs could get reordered in the network. Therefore, the
final state is inferred from the presence or absence of the
MP_CAPABLE option in the third packet of the TCP handshake. If this
option is not present, the connection SHOULD fall back to regular
TCP, as documented in Section 3.6.

The initial data sequence number on an MPTCP connection is generated
from the key. The algorithm for IDSN generation is also determined
from the negotiated authentication algorithm. In this specification,
with only the SHA-1 algorithm specified and selected, the IDSN of a
host MUST be the least significant 64 bits of the SHA-1 hash of its
key, i.e., IDSN-A = Hash(Key-A) and IDSN-B = Hash(Key-B). This
deterministic generation of the IDSN allows a receiver to ensure that
there are no gaps in sequence space at the start of the connection.
The SYN with MP_CAPABLE occupies the first octet of data sequence
space, although this does not need to be acknowledged at the
connection level until the first data is sent (see Section 3.3).

3.2. Starting a New Subflow

   Once an MPTCP connection has begun with the MP_CAPABLE exchange,
   further subflows can be added to the connection. Hosts have
   knowledge of their own address(es), and can become aware of the other
   host's addresses through signaling exchanges as described in
   Section 3.4. Using this knowledge, a host can initiate a new subflow
   over a currently unused pair of addresses. It is permitted for
   either host in a connection to initiate the creation of a new
   subflow, but it is expected that this will normally be the original
   connection initiator (see Section 3.8 for heuristics).

   A new subflow is started as a normal TCP SYN/ACK exchange. The Join
   Connection (MP_JOIN) TCP option is used to identify the connection to
   be joined by the new subflow. It uses keying material that was
   exchanged in the initial MP_CAPABLE handshake (Section 3.1), and that
   handshake also negotiates the crypto algorithm in use for the MP_JOIN
   handshake.

   This section specifies the behavior of MP_JOIN using the HMAC-SHA1
   algorithm. An MP_JOIN option is present in the SYN, SYN/ACK, and ACK
   of the three-way handshake, although in each case with a different
   format.

   In the first MP_JOIN on the SYN packet, illustrated in Figure 5, the
   initiator sends a token, random number, and address ID.

   The token is used to identify the MPTCP connection and is a
   cryptographic hash of the receiver's key, as exchanged in the initial
   MP_CAPABLE handshake (Section 3.1). In this specification, the
   tokens presented in this option are generated by the SHA-1 ([4],
   [15]) algorithm, truncated to the most significant 32 bits. The
   token included in the MP_JOIN option is the token that the receiver
   of the packet uses to identify this connection; i.e., Host A will

send Token-B (which is generated from Key-B). Note that the hash
generation algorithm can be overridden by the choice of cryptographic
handshake algorithm, as defined in Section 3.1.

The MP_JOIN SYN sends not only the token (which is static for a
connection) but also random numbers (nonces) that are used to prevent
replay attacks on the authentication method. Recommendations for the
generation of random numbers for this purpose are given in [14].

The MP_JOIN option includes an "Address ID". This is an identifier
that only has significance within a single connection, where it
identifies the source address of this packet, even if the IP header
has been changed in transit by a middlebox. The Address ID allows
address removal (Section 3.4.2) without needing to know what the
source address at the receiver is, thus allowing address removal
through NATs. The Address ID also allows correlation between new
subflow setup attempts and address signaling (Section 3.4.1), to
prevent setting up duplicate subflows on the same path, if an MP_JOIN
and ADD_ADDR are sent at the same time.

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. A host MUST store the mappings between Address IDs and
addresses both for itself and the remote host. An implementation
will also need to know which local and remote Address IDs are
associated with which established subflows, for when addresses are
removed from a local or remote host.

The MP_JOIN option on packets with the SYN flag set also includes 4
bits of flags, 3 of which are currently reserved and MUST be set to
zero by the sender. The final bit, labeled "B", indicates whether
the sender of this option wishes this subflow to be used as a backup
path (B=1) in the event of failure of other paths, or whether it
wants it to be used as part of the connection immediately. By
setting B=1, the sender of the option is requesting the other host to
only send data on this subflow if there are no available subflows
where B=0. Subflow policy is discussed in more detail in
Section 3.3.8.

```
                        1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +---------------+---------------+-------+-----+-+---------------+
   |     Kind      |  Length = 12  |Subtype|     |B|  Address ID   |
   +---------------+---------------+-------+-----+-+---------------+
   |                   Receiver's Token (32 bits)                  |
   +---------------------------------------------------------------+
   |                Sender's Random Number (32 bits)               |
   +---------------------------------------------------------------+
```
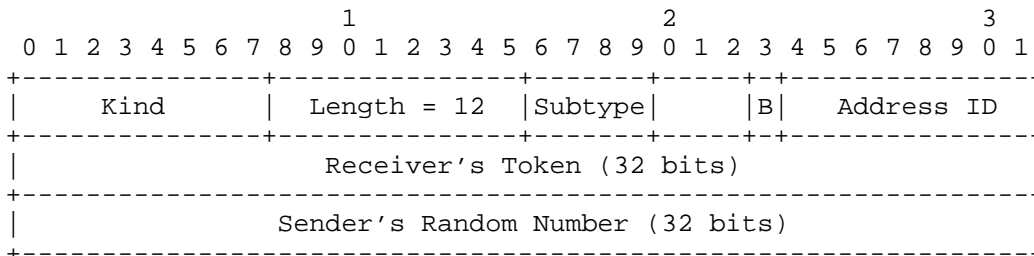
       Figure 5: Join Connection (MP_JOIN) Option (for Initial SYN)

   When receiving a SYN with an MP_JOIN option that contains a valid
   token for an existing MPTCP connection, the recipient SHOULD respond
   with a SYN/ACK also containing an MP_JOIN option containing a random
   number and a truncated (leftmost 64 bits) Hash-based Message
   Authentication Code (HMAC). This version of the option is shown in
   Figure 6. If the token is unknown, or the host wants to refuse
   subflow establishment (for example, due to a limit on the number of
   subflows it will permit), the receiver will send back a reset (RST)
   signal, analogous to an unknown port in TCP. Although calculating an
   HMAC requires cryptographic operations, it is believed that the 32-
   bit token in the MP_JOIN SYN gives sufficient protection against
   blind state exhaustion attacks; therefore, there is no need to
   provide mechanisms to allow a responder to operate statelessly at the
   MP_JOIN stage.

   An HMAC is sent by both hosts -- by the initiator (Host A) in the
   third packet (the ACK) and by the responder (Host B) in the second
   packet (the SYN/ACK). Doing the HMAC exchange at this stage allows
   both hosts to have first exchanged random data (in the first two SYN
   packets) that is used as the "message". This specification defines
   that HMAC as defined in [10] is used, along with the SHA-1 hash
   algorithm [4] (potentially implemented as in [15]), thus generating a
   160-bit / 20-octet HMAC. Due to option space limitations, the HMAC
   included in the SYN/ACK is truncated to the leftmost 64 bits, but
   this is acceptable since random numbers are used; thus, an attacker
   only has one chance to guess the HMAC correctly (if the HMAC is
   incorrect, the TCP connection is closed, so a new MP_JOIN negotiation
   with a new random number is required).

   The initiator's authentication information is sent in its first ACK
   (the third packet of the handshake), as shown in Figure 7. This data
   needs to be sent reliably, since it is the only time this HMAC is
   sent; therefore, receipt of this packet MUST trigger a regular TCP
   ACK in response, and the packet MUST be retransmitted if this ACK is
   not received. In other words, sending the ACK/MP_JOIN packet places
   the subflow in the PRE_ESTABLISHED state, and it moves to the

ESTABLISHED state only on receipt of an ACK from the receiver. It is
not permitted to send data while in the PRE_ESTABLISHED state. The
reserved bits in this option MUST be set to zero by the sender.

The key for the HMAC algorithm, in the case of the message
transmitted by Host A, will be Key-A followed by Key-B, and in the
case of Host B, Key-B followed by Key-A. These are the keys that
were exchanged in the original MP_CAPABLE handshake. The "message"
for the HMAC algorithm in each case is the concatenations of random
number for each host (denoted by R): for Host A, R-A followed by R-B;
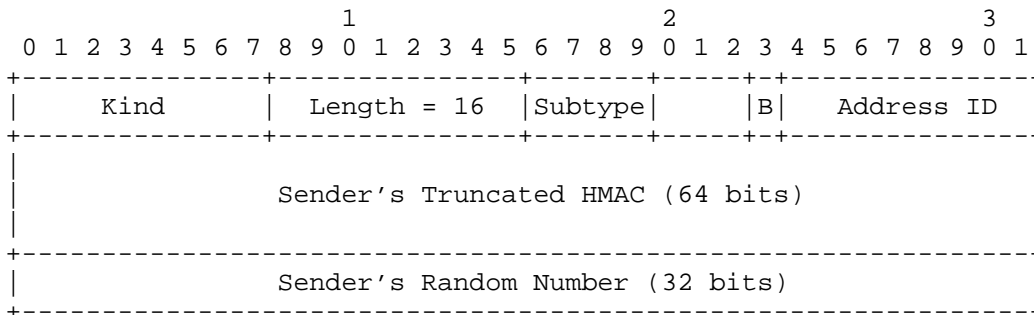and for Host B, R-B followed by R-A.

```
                        1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +---------------+---------------+-------+-----+-+---------------+
   |     Kind      |  Length = 16  |Subtype|     |B|  Address ID   |
   +---------------+---------------+-------+-----+-+---------------+
   |                                                               |
   |                Sender's Truncated HMAC (64 bits)              |
   |                                                               |
   +---------------------------------------------------------------+
   |                Sender's Random Number (32 bits)               |
   +---------------------------------------------------------------+
```

   Figure 6: Join Connection (MP_JOIN) Option (for Responding SYN/ACK)

```
                        1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +---------------+---------------+-------+-----------------------+
   |     Kind      |  Length = 24  |Subtype|      (reserved)       |
   +---------------+---------------+-------+-----------------------+
   |                                                               |
   |                                                               |
   |                   Sender's HMAC (160 bits)                    |
   |                                                               |
   |                                                               |
   +---------------------------------------------------------------+
```
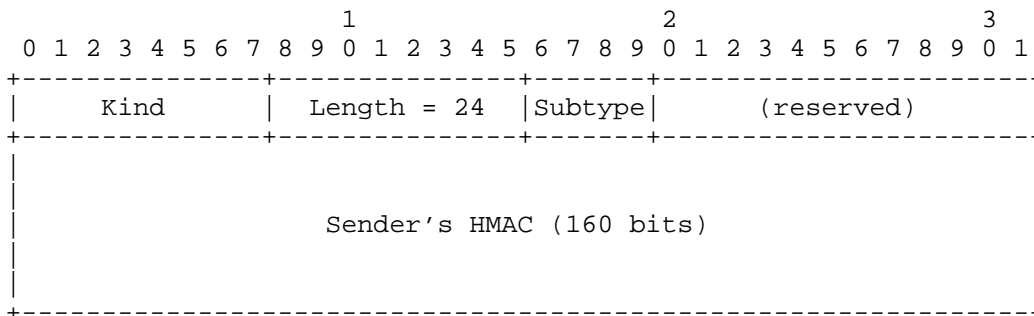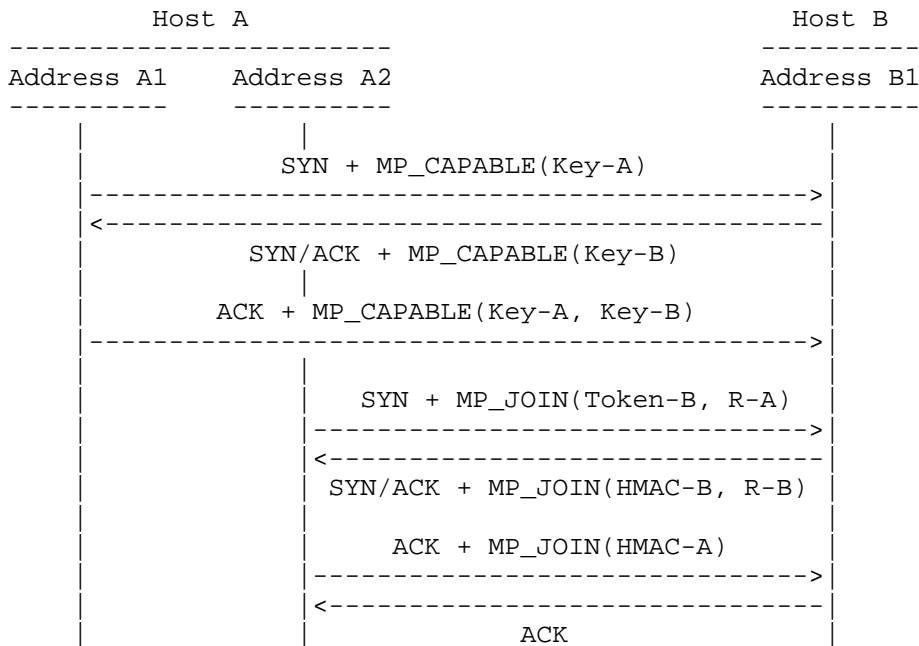
      Figure 7: Join Connection (MP_JOIN) Option (for Third ACK)

These various TCP options fit together to enable authenticated
subflow setup as illustrated in Figure 8.

```
              Host A                                   Host B
      -----------------------                        ----------
   Address A1      Address A2                        Address B1
   ----------      ----------                        ----------
       |               |                                 |
       |               |     SYN + MP_CAPABLE(Key-A)     |
       |------------------------------------------------>|
       |<------------------------------------------------|
       |               |   SYN/ACK + MP_CAPABLE(Key-B)   |
       |               |                                 |
       |               |   ACK + MP_CAPABLE(Key-A, Key-B)|
       |------------------------------------------------>|
       |               |                                 |
       |               |      SYN + MP_JOIN(Token-B, R-A)|
       |               |-------------------------------->|
       |               |<--------------------------------|
       |               |  SYN/ACK + MP_JOIN(HMAC-B, R-B) |
       |               |                                 |
       |               |      ACK + MP_JOIN(HMAC-A)      |
       |               |-------------------------------->|
       |               |<--------------------------------|
       |               |              ACK                |
```

    HMAC-A = HMAC(Key=(Key-A+Key-B), Msg=(R-A+R-B))
    HMAC-B = HMAC(Key=(Key-B+Key-A), Msg=(R-B+R-A))

              Figure 8: Example Use of MPTCP Authentication

   If the token received at Host B is unknown or local policy prohibits
   the acceptance of the new subflow, the recipient MUST respond with a
   TCP RST for the subflow.

   If the token is accepted at Host B, but the HMAC returned to Host A
   does not match the one expected, Host A MUST close the subflow with a
   TCP RST.

   If Host B does not receive the expected HMAC, or the MP_JOIN option
   is missing from the ACK, it MUST close the subflow with a TCP RST.

   If the HMACs are verified as correct, then both hosts have
   authenticated each other as being the same peers as existed at the
   start of the connection, and they have agreed of which connection
   this subflow will become a part.

   If the SYN/ACK as received at Host A does not have an MP_JOIN option,
   Host A MUST close the subflow with a RST.

This covers all cases of the loss of an MP_JOIN. In more detail, if
MP_JOIN is stripped from the SYN on the path from A to B, and Host B
does not have a passive opener on the relevant port, it will respond
with a RST in the normal way. If in response to a SYN with an
MP_JOIN option, a SYN/ACK is received without the MP_JOIN option
(either since it was stripped on the return path, or it was stripped
on the outgoing path but the passive opener on Host B responded as if
it were a new regular TCP session), then the subflow is unusable and
Host A MUST close it with a RST.

Note that additional subflows can be created between any pair of
ports (but see Section 3.8 for heuristics); no explicit application-
level accept calls or bind calls are required to open additional
subflows. To associate a new subflow with an existing connection,
the token supplied in the subflow's SYN exchange is used for
demultiplexing. This then binds the 5-tuple of the TCP subflow to
the local token of the connection. A consequence is that it is
possible to allow any port pairs to be used for a connection.

Demultiplexing 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 set up, demultiplexing
packets is done using the 5-tuple, as in traditional TCP. The
5-tuples will be mapped to the local connection identifier (token).
Note that Host A will know its local token for the subflow even
though it is not sent on the wire -- only the responder's token is
sent.

3.3. General MPTCP Operation

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 behavior in detail.

The data sequence mapping and the Data ACK are signaled in the Data
Sequence Signal (DSS) option (Figure 9). Either or both can be
signaled in one DSS, dependent on the flags set. The data sequence
mapping defines how the sequence space on the subflow maps to the
connection level, and the Data ACK acknowledges receipt of data at
the connection level. These functions are described in more detail
in the following two subsections.

Either or both the data sequence mapping and the Data ACK can be
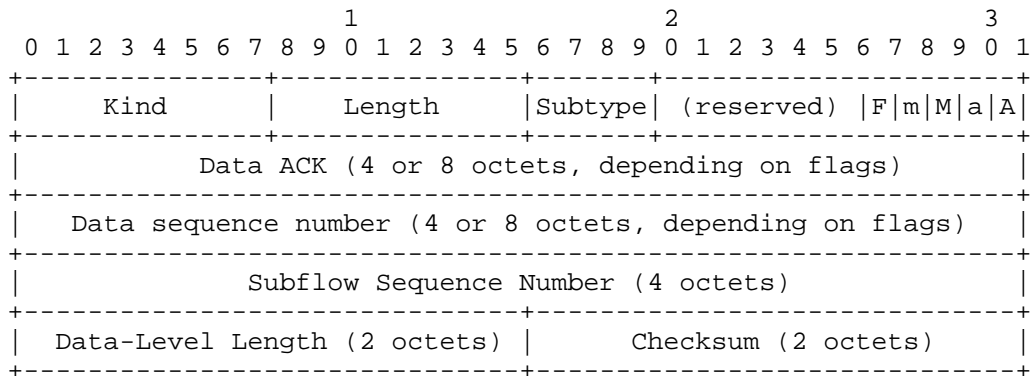signaled in the DSS option, dependent on the flags set.

```
                         1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +---------------+---------------+-------+----------------------+
    |     Kind      |     Length    |Subtype| (reserved) |F|m|M|a|A|
    +---------------+---------------+-------+----------------------+
    |           Data ACK (4 or 8 octets, depending on flags)       |
    +--------------------------------------------------------------+
    |   Data sequence number (4 or 8 octets, depending on flags)   |
    +--------------------------------------------------------------+
    |              Subflow Sequence Number (4 octets)              |
    +-------------------------------+------------------------------+
    |  Data-Level Length (2 octets) |      Checksum (2 octets)     |
    +-------------------------------+------------------------------+
```

                 Figure 9: Data Sequence Signal (DSS) Option

   The flags, when set, define the contents of this option, as follows:

   o A = Data ACK present

   o a = Data ACK is 8 octets (if not set, Data ACK is 4 octets)

   o M = Data Sequence Number (DSN), Subflow Sequence Number (SSN),
      Data-Level Length, and Checksum present

   o m = Data sequence number is 8 octets (if not set, DSN is 4 octets)

   The flags 'a' and 'm' only have meaning if the corresponding 'A' or
   'M' flags are set; otherwise, they will be ignored. The maximum
   length of this option, with all flags set, is 28 octets.

   The 'F' flag indicates "DATA_FIN". If present, this means that this
   mapping covers the final data from the sender. This is the
   connection-level equivalent to the FIN flag in single-path TCP. A
   connection is not closed unless there has been a DATA_FIN exchange or
   a timeout. The purpose of the DATA_FIN and the interactions between
   this flag, the subflow-level FIN flag, and the data sequence mapping
   are described in Section 3.3.3. The remaining reserved bits MUST be
   set to zero by an implementation of this specification.

   Note that the checksum is only present in this option if the use of
   MPTCP checksumming has been negotiated at the MP_CAPABLE handshake
   (see Section 3.1). The presence of the checksum can be inferred from
   the length of the option. If a checksum is present, but its use had
   not been negotiated in the MP_CAPABLE handshake, the checksum field
   MUST be ignored. If a checksum is not present when its use has been
   negotiated, the receiver MUST close the subflow with a RST as it is
   considered broken.

3.3.1. Data Sequence Mapping

   The data stream as a whole can be reassembled through the use of the
   data sequence mapping components of the DSS option (Figure 9), which
   define the mapping from the subflow sequence number to the data
   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 [11] is used at the subflow level to improve
   efficiency.

   The data sequence mapping specifies a mapping from subflow sequence
   space to data sequence space. This is expressed in terms of starting
   sequence numbers for the subflow and the data level, and a length of
   bytes for which this mapping is valid. This explicit mapping for a
   range of data was chosen rather than per-packet signaling 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). It also allows a single mapping to cover many
   packets, which may be useful in bulk transfer situations.

   A mapping is fixed, in that the subflow sequence number is bound to
   the data sequence number after the mapping has been processed. A
   sender MUST NOT change this mapping after it has been declared;
   however, the same data sequence number can be mapped to by different
   subflows for retransmission purposes (see Section 3.3.6). This would
   also permit the same data to be sent simultaneously on multiple
   subflows for resilience or efficiency purposes, especially in the
   case of lossy links. Although the detailed specification of such
   operation is outside the scope of this document, an implementation
   SHOULD treat the first data that is received at a subflow for the
   data sequence space as that which should be delivered to the
   application, and any later data for that sequence space ignored.

   The data sequence number is specified as an absolute value, whereas
   the subflow sequence numbering is relative (the SYN at the start of
   the subflow has relative subflow sequence number 0). This is to
   allow middleboxes to change the initial sequence number of a subflow,
   such as firewalls that undertake ISN randomization.

   The data sequence mapping also contains a checksum of the data that
   this mapping covers, if use of checksums has been negotiated at the
   MP_CAPABLE exchange. Checksums are used to detect if the payload has
   been adjusted in any way by a non-MPTCP-aware middlebox. If this
   checksum fails, it will trigger a failure of the subflow, or a

fallback to regular TCP, as documented in Section 3.6, since MPTCP
can no longer reliably know the subflow sequence space at the
receiver to build data sequence mappings.

The checksum algorithm used is the standard TCP checksum [1],
operating over the data covered by this mapping, along with a pseudo-
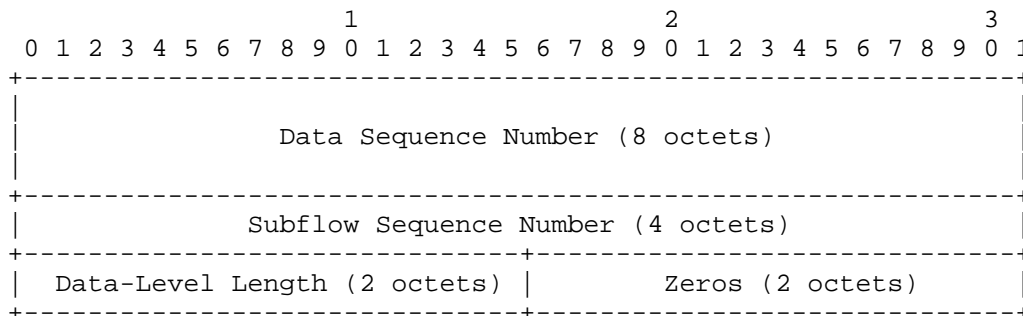header as shown in Figure 10.

```
                            1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +--------------------------------------------------------------+
   |                                                              |
   |                                                              |
   |                Data Sequence Number (8 octets)               |
   |                                                              |
   |                                                              |
   +--------------------------------------------------------------+
   |                Subflow Sequence Number (4 octets)            |
   +-----------------------------+--------------------------------+
   |  Data-Level Length (2 octets) |        Zeros (2 octets)      |
   +-----------------------------+--------------------------------+
```

             Figure 10: Pseudo-Header for DSS Checksum

Note that the data sequence number used in the pseudo-header is
always the 64-bit value, irrespective of what length is used in the
DSS option itself. The standard TCP checksum algorithm has been
chosen since it will be calculated anyway for the TCP subflow, and if
calculated first over the data before adding the pseudo-headers, it
only needs to be calculated once. Furthermore, since the TCP
checksum is additive, the checksum for a DSN_MAP can be constructed
by simply adding together the checksums for the data of each
constituent TCP segment, and adding the checksum for the DSS pseudo-
header.

Note that checksumming relies on the TCP subflow containing
contiguous data; therefore, a TCP subflow MUST NOT use the Urgent
Pointer to interrupt an existing mapping. Further note, however,
that if Urgent data is received on a subflow, it SHOULD be mapped to
the data sequence space and delivered to the application analogous to
Urgent data in regular TCP.

To avoid possible deadlock scenarios, subflow-level processing should
be undertaken separately from that at connection level. Therefore,
even if a mapping does not exist from the subflow space to the data-
level space, the data SHOULD still be ACKed at the subflow (if it is
in-window). This data cannot, however, be acknowledged at the data
level (Section 3.3.2) because its data sequence numbers are unknown.
Implementations MAY hold onto such unmapped data for a short while in
the expectation that a mapping will arrive shortly. Such unmapped

data cannot be counted as being within the connection level receive
window because this is relative to the data sequence numbers, so if
the receiver runs out of memory to hold this data, it will have to be
discarded. If a mapping for that subflow-level sequence space does
not arrive within a receive window of data, that subflow SHOULD be
treated as broken, closed with a RST, and any unmapped data silently
discarded.

Data sequence numbers are always 64-bit quantities, and MUST be
maintained as such in implementations. If a connection is
progressing at a slow rate, so protection against wrapped sequence
numbers is not required, then it is permissible to include just the
lower 32 bits of the data sequence number in the data sequence
mapping and/or Data ACK as an optimization, and an implementation can
make this choice independently for each packet.

An implementation MUST send the full 64-bit data sequence number if
it is transmitting at a sufficiently high rate that the 32-bit value
could wrap within the Maximum Segment Lifetime (MSL) [16]. The
lengths of the DSNs used in these values (which may be different) are
declared with flags in the DSS option. Implementations MUST accept a
32-bit DSN and implicitly promote it to a 64-bit quantity by
incrementing the upper 32 bits of sequence number each time the lower
32 bits wrap. A sanity check MUST be implemented to ensure that a
wrap occurs at an expected time (e.g., the sequence number jumps from
a very high number to a very low number) and is not triggered by out-
of-order packets.

As with the standard TCP sequence number, the data sequence number
should not start at zero, but at a random value to make blind session
hijacking harder. This specification requires setting the initial
data sequence number (IDSN) of each host to the least significant 64
bits of the SHA-1 hash of the host's key, as described in
Section 3.1.

A 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 the 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 hosts,
another is when segments of data are scheduled in larger than packet-
sized chunks.

An "infinite" mapping can be used to fall back to regular TCP by
mapping the subflow-level data to the connection-level data for the
remainder of the connection (see Section 3.6). This is achieved by
setting the Data-Level Length field of the DSS option to the reserved
value of 0. The checksum, in such a case, will also be set to zero.

3.3.2. Data Acknowledgments

   To provide full end-to-end resilience, MPTCP provides a connection-
   level acknowledgment, to act as a cumulative ACK for the connection
   as a whole. This is the "Data ACK" field of the DSS option
   (Figure 9). The Data ACK is analogous to the behavior of the
   standard TCP cumulative ACK -- indicating how much data has been
   successfully received (with no holes). This is in comparison to the
   subflow-level ACK, which acts analogous to TCP SACK, given that there
   may still be holes in the data stream at the connection level. The
   Data ACK specifies the next data sequence number it expects to
   receive.

   The Data ACK, as for the DSN, can be sent as the full 64-bit value,
   or as the lower 32 bits. If data is received with a 64-bit DSN, it
   MUST be acknowledged with a 64-bit Data ACK. If the DSN received is
   32 bits, it is valid for the implementation to choose whether to send
   a 32-bit or 64-bit Data ACK.

   The Data ACK proves that the data, and all required MPTCP signaling,
   has been received and accepted by the remote end. One key use of the
   Data ACK signal is that it is used to indicate the left edge of the
   advertised receive window. As explained in Section 3.3.4, the
   receive window is shared by all subflows and is relative to the Data
   ACK. Because of this, an implementation MUST NOT use the RCV.WND
   field of a TCP segment at the connection level if it does not also
   carry a DSS option with a Data ACK field. Furthermore, separating
   the connection-level acknowledgments from the subflow level allows
   processing to be done separately, and a receiver has the freedom to
   drop segments after acknowledgment at the subflow level, for example,
   due to memory constraints when many segments arrive out of order.

   An MPTCP sender MUST NOT free data from the send buffer until it has
   been acknowledged by both a Data ACK received on any subflow and at
   the subflow level by all subflows on which the data was sent. The
   former condition ensures liveness of the connection and the latter
   condition ensures liveness and self-consistence of a subflow when
   data needs to be retransmitted. Note, however, that if some data
   needs to be retransmitted multiple times over a subflow, there is a
   risk of blocking the sending window. In this case, the MPTCP sender
   can decide to terminate the subflow that is behaving badly by sending
   a RST.

   The Data ACK MAY be included in all segments; however, optimizations
   SHOULD be considered in more advanced implementations, where the Data
   ACK is present in segments only when the Data ACK value advances, and

this behavior MUST be treated as valid. This behavior ensures the
sender buffer is freed, while reducing overhead when the data
transfer is unidirectional.

3.3.3. Closing a Connection

In regular TCP, a FIN announces the receiver that the sender has no
more data to send. In order to allow subflows to operate
independently and to keep the appearance of TCP over the wire, 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; for regular TCP, this would result in a FIN
on the connection. For MPTCP, an equivalent mechanism is needed, and
this is referred to as the DATA_FIN.

A DATA_FIN is an indication that the sender has no more data to send,
and as such can be used to verify that all data has been successfully
received. A DATA_FIN, as with the FIN on a regular TCP connection,
is a unidirectional signal.

The DATA_FIN is signaled by setting the 'F' flag in the Data Sequence
Signal option (Figure 9) to 1. A DATA_FIN occupies 1 octet (the
final octet) of the connection-level sequence space. Note that the
DATA_FIN is included in the Data-Level Length, but not at the subflow
level: for example, a segment with DSN 80, and Data-Level Length 11,
with DATA_FIN set, would map 10 octets from the subflow into data
sequence space 80-89, the DATA_FIN is DSN 90; therefore, this segment
including DATA_FIN would be acknowledged with a DATA_ACK of 91.

Note that when the DATA_FIN is not attached to a TCP segment
containing data, the Data Sequence Signal MUST have a subflow
sequence number of 0, a Data-Level Length of 1, and the data sequence
number that corresponds with the DATA_FIN itself. The checksum in
this case will only cover the pseudo-header.

A DATA_FIN has the semantics and behavior as a regular TCP FIN, but
at the connection level. Notably, it is only DATA_ACKed once all
data has been successfully received at the connection level. Note,
therefore, that a DATA_FIN is decoupled from a subflow FIN. It is
only permissible to combine these signals on one subflow if there is
no data outstanding on other subflows. Otherwise, it may be
necessary to retransmit data on different subflows. Essentially, a
host MUST NOT close all functioning subflows unless it is safe to do

so, i.e., until all outstanding data has been DATA_ACKed, or until
the segment with the DATA_FIN flag set is the only outstanding
segment.

Once a DATA_FIN has been acknowledged, all remaining subflows MUST be
closed with standard FIN exchanges. Both hosts SHOULD send FINs on
all subflows, as a courtesy to allow middleboxes to clean up state
even if an individual subflow has failed. It is also encouraged to
reduce the timeouts (Maximum Segment Life) on subflows at end hosts.
In particular, any subflows where there is still outstanding data
queued (which has been retransmitted on other subflows in order to
get the DATA_FIN acknowledged) MAY be closed with a RST.

A connection is considered closed once both hosts' DATA_FINs have
been acknowledged by DATA_ACKs.

As specified above, a standard TCP FIN on an individual subflow only
shuts down the subflow on which it was sent. If all subflows have
been closed with a FIN exchange, but no DATA_FIN has been received
and acknowledged, the MPTCP connection is treated as closed only
after a timeout. This implies that an implementation will have
TIME_WAIT states at both the subflow and connection levels (see
Appendix C). This permits "break-before-make" scenarios where
connectivity is lost on all subflows before a new one can be re-
established.

3.3.4. 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 (i.e.,
DATA_ACK + receive window). The receiver will use the data sequence
number to tell if a packet should be accepted at the connection
level.

When deciding to accept packets at subflow level, regular TCP checks
the sequence number in the packet against the allowed receive window.
With multipath, such a check is done using only the connection-level

window. A sanity check SHOULD be performed at subflow level to
ensure that the subflow and mapped sequence numbers meet the
following test: SSN - SUBFLOW_ACK <= DSN - DATA_ACK, where SSN is the
subflow sequence number of the received packet and SUBFLOW_ACK is the
RCV.NXT (next expected sequence number) of the subflow (with the
equivalent connection-level definitions for DSN and DATA_ACK).

In regular TCP, once a segment is deemed in-window, it is put either
in the in-order receive queue or in the out-of-order queue. In
Multipath TCP, the same happens but at the connection level: a
segment is placed in the connection level in-order or out-of-order
queue if it is in-window at both connection and subflow levels. The
stack still has to remember, for each subflow, which segments were
received successfully so that it can ACK them at subflow level
appropriately. Typically, this will be implemented by keeping per
subflow out-of-order queues (containing only message headers, not the
payloads) and remembering the value of the cumulative ACK.

It is important for implementers to understand how large a receiver
buffer is appropriate. The lower bound for full network utilization
is the maximum bandwidth-delay product of any one of the paths.
However, this might be insufficient when a packet is lost on a slower
subflow and needs to be retransmitted (see Section 3.3.6). A tight
upper bound would be the maximum round-trip time (RTT) of any path
multiplied by the total bandwidth available across all paths. This
permits all subflows to continue at full speed while a packet is
fast-retransmitted on the maximum RTT path. Even this might be
insufficient to maintain full performance in the event of a
retransmit timeout on the maximum RTT path. It is for future study
to determine the relationship between retransmission strategies and
receive buffer sizing.

3.3.5. Sender Considerations

The sender remembers receiver window advertisements from the
receiver. It should only update its local receive window values when
the largest sequence number allowed (i.e., DATA_ACK + receive window)
increases, on the receipt of a DATA_ACK. This is important to allow
using paths with different RTTs, and thus different feedback loops.

MPTCP uses a single receive window across all subflows, and if the
receive window was guaranteed to be unchanged end-to-end, a host
could always read the most recent receive window value. However,
some classes of middleboxes may alter the TCP-level receive window.
Typically, these will shrink the offered window, although for short
periods of time it may be possible for the window to be larger
(however, note that this would not continue for long periods since
ultimately the middlebox must keep up with delivering data to the

receiver). Therefore, if receive window sizes differ on multiple
subflows, when sending data MPTCP SHOULD take the largest of the most
recent window sizes as the one to use in calculations. This rule is
implicit in the requirement not to reduce the right edge of the
window.

The sender MUST also remember the receive windows advertised by each
subflow. The allowed window for subflow i is (ack_i, ack_i +
rcv_wnd_i), where ack_i is the subflow-level cumulative ACK of
subflow i. This ensures data will not be sent to a middlebox unless
there is enough buffering for the data.

Putting the two rules together, we get the following: a sender is
allowed to send data segments with data-level sequence numbers
between (DATA_ACK, DATA_ACK + receive_window). Each of these
segments will be mapped onto subflows, as long as subflow sequence
numbers are in the allowed windows for those subflows. Note that
subflow sequence numbers do not generally affect flow control if the
same receive window is advertised across all subflows. They will
perform flow control for those subflows with a smaller advertised
receive window.

The send buffer MUST, at a minimum, be as big as the receive buffer,
to enable the sender to reach maximum throughput.

3.3.6. Reliability and Retransmissions

The data sequence mapping allows senders to resend data with the same
data sequence number on a different subflow. When doing this, a host
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 sensible behavior. Optimizations 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 3.3.8). 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.

It is envisaged that a standard connection-level retransmission
mechanism would be implemented around a connection-level data queue:
all segments that haven't been DATA_ACKed are stored. A timer is set
when the head of the connection-level is ACKed at subflow level but
its corresponding data is not ACKed at data level. This timer will
guard against failures in retransmission by middleboxes that
proactively ACK data.

The sender MUST keep data in its send buffer as long as the data has
not been acknowledged at both connection level and on all subflows on
which it has been sent. 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 after a timeout,
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 (which MAY be lower than
the usual TCP limits of the Maximum Segment Life), or on the receipt
of an ICMP error, and only then delete the outstanding data segments.

Multiple retransmissions are triggers that will indicate that a
subflow performs badly and could lead to a host resetting the subflow
with a RST. However, additional research is required to understand
the heuristics of how and when to reset underperforming subflows.
For example, a highly asymmetric path may be misdiagnosed as
underperforming.

3.3.7. Congestion Control Considerations

Different subflows in an MPTCP connection have different congestion
windows. To achieve fairness at bottlenecks and 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 [5]; the algorithm does not
achieve perfect resource pooling but is "safe" in that it is readily
deployable in the current Internet. By this, we mean that it does
not take up more capacity on any one path than if it was a single
path flow using only that route, so this ensures fair coexistence
with single-path TCP at shared bottlenecks.

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, as well as
those for achieving different properties in quality of service,
reliability, and resilience.

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 were lost and when.

3.3.8. Subflow Policy

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 maximize throughput,
all available paths will be used simultaneously for data transfer,
using coupled congestion control as described in [5]. 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 (of delay or bandwidth) is more important than
throughput. Application requirements such as these are discussed in
detail in [6].

The ability to make effective choices at the sender requires full
knowledge of the path "cost", which is unlikely to be the case. It
would be desirable for a receiver to be able to signal their own
preferences for paths, since they will often be the multihomed party,
and may have to pay for metered incoming bandwidth.

Whilst fine-grained control may be the most powerful solution, that
would require some mechanism such as overloading the Explicit
Congestion Notification (ECN) signal [17], which is undesirable, and
it is felt that there would not be sufficient benefit to justify an
entirely new signal. Therefore, the MP_JOIN option (see Section 3.2)
contains the 'B' bit, which allows a host to indicate to its peer
that this path should be treated as a backup path to use only in the
event of failure of other working subflows (i.e., a subflow where the
receiver has indicated B=1 SHOULD NOT be used to send data unless
there are no usable subflows where B=0).

In the event that the available set of paths changes, a host may wish
to signal a change in priority of subflows to the peer (e.g., a
subflow that was previously set as backup should now take priority

over all remaining subflows). Therefore, the MP_PRIO option, shown
in Figure 11, can be used to change the 'B' flag of the subflow on
which it is sent.

```
                        1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +---------------+---------------+-------+-----+-+-------------+
   |     Kind      |    Length     |Subtype|     |B| AddrID (opt) |
   +---------------+---------------+-------+-----+-+-------------+
```
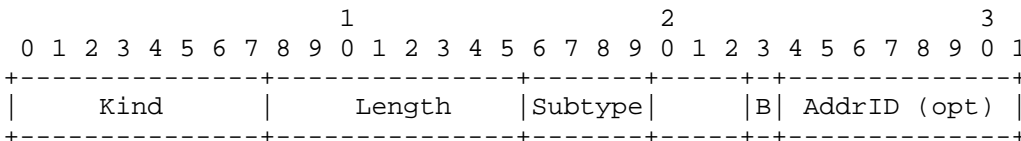
            Figure 11: Change Subflow Priority (MP_PRIO) Option

It should be noted that the backup flag is a request from a data
receiver to a data sender only, and the data sender SHOULD adhere to
these requests. A host cannot assume that the data sender will do
so, however, since local policies -- or technical difficulties -- may
override MP_PRIO requests. Note also that this signal applies to a
single direction, and so the sender of this option could choose to
continue using the subflow to send data even if it has signaled B=1
to the other host.

This option can also be applied to other subflows than the one on
which it is sent, by setting the optional Address ID field. This
applies the given setting of B to all subflows in this connection
that use the address identified by the given Address ID. The
presence of this field is determined by the option length; if
Length==4 then it is present. If Length==3, then it applies to the
current subflow only. The use case of this is that a host can signal
to its peer that an address is temporarily unavailable (for example,
if it has radio coverage issues) and the peer should therefore drop
to backup state on all subflows using that Address ID.

3.4. Address Knowledge Exchange (Path Management)

We use the term "path management" to refer to the exchange of
information about additional paths between hosts, which in this
design is managed by multiple addresses at hosts. For more detail of
the architectural thinking behind this design, see the MPTCP
Architecture document [2].

This design makes use of two methods of sharing such information, and
both can be used on a connection. The first is the direct setup of
new subflows, already described in Section 3.2, where the initiator
has an additional address. The second method, described in the
following subsections, signals addresses explicitly to the other host
to allow it to initiate new subflows. The two mechanisms are
complementary: the first is implicit and simple, while the explicit
is more complex but is more robust. Together, the mechanisms allow

   addresses to change in flight (and thus support operation through
   NATs, since the source address need not be known), and also allow the
   signaling of previously unknown addresses, and of addresses belonging
   to other address families (e.g., both IPv4 and IPv6).

   Here is an example of typical operation of the protocol:

   o An MPTCP connection is initially set up between address/port A1 of
      Host A and address/port B1 of Host B. If Host A is multihomed and
      multiaddressed, it can start an additional subflow from its
      address A2 to B1, by sending a SYN with a Join option from A2 to
      B1, using B's previously declared token for this connection.
      Alternatively, if B is multihomed, it can try to set up a new
      subflow from B2 to A1, using A's previously declared token. In
      either case, the SYN will be sent to the port already in use for
      the original subflow on the receiving host.

   o Simultaneously (or after a timeout), an ADD_ADDR option
      (Section 3.4.1) 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. In our example, A will send ADD_ADDR option
      informing B of address/port A2. The mix of using the SYN-based
      option and the ADD_ADDR option, including timeouts, is
      implementation specific and can be tailored to agree with local
      policy.

   o If subflow A2-B1 is successfully set up, Host B can use the
      Address ID in the Join option to correlate this with the ADD_ADDR
      option that will also arrive on an existing subflow; now B knows
      not to open A2-B1, ignoring the ADD_ADDR. Otherwise, if B has not
      received the A2-B1 MP_JOIN SYN but received the ADD_ADDR, it can
      try to initiate a new subflow from one or more of its addresses to
      address A2. This permits new sessions to be opened if one host is
      behind a NAT.

   Other ways of using the two signaling mechanisms are possible; for
   instance, signaling addresses in other address families can only be
   done explicitly using the Add Address option.

3.4.1. Address Advertisement

   The Add Address (ADD_ADDR) TCP option announces additional addresses
   (and optionally, ports) on which a host can be reached (Figure 12).
   Multiple instances of this TCP option can be added in a single
   message 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.
As with all MPTCP signals, the receiver MUST undertake standard TCP
validity checks before acting upon it.

Every address has an Address ID that can be used for uniquely
identifying the address within a connection for address removal.
This is also used to identify MP_JOIN options (see Section 3.2)
relating to the same address, even when address translators are in
use. The Address ID MUST uniquely identify the address to the sender
(within the scope of the connection), but the mechanism for
allocating such IDs is implementation specific.

All address IDs learned via either MP_JOIN or ADD_ADDR SHOULD 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. Note that an implementation
MAY discard incoming address advertisements at will, for example, for
avoiding the required mapping state, or because advertised addresses
are of no use to it (for example, IPv6 addresses when it has IPv4
only). Therefore, a host MUST treat address advertisements as soft
state, and it MAY choose to refresh advertisements periodically.

This option is shown in Figure 12. The illustration is sized for
IPv4 addresses (IPVer = 4). For IPv6, the IPVer field will read 6,
and the length of the address will be 16 octets (instead of 4).

The presence of the final 2 octets, specifying the TCP port number to
use, are optional and can be inferred from the length of the option.
Although it is expected that the majority of use cases will use the
same port pairs as used for the initial subflow (e.g., port 80
remains port 80 on all subflows, as does the ephemeral port at the
client), there may be cases (such as port-based load balancing) where
the explicit specification of a different port is required. If no
port is specified, MPTCP SHOULD attempt to connect to the specified
address on the same port as is already in use by the subflow on which
the ADD_ADDR signal was sent; this is discussed in more detail in
Section 3.8.

```
                    1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-------+---------------+
|     Kind      |    Length     |Subtype| IPVer |  Address ID   |
+---------------+---------------+-------+-------+---------------+
|          Address (IPv4 - 4 octets / IPv6 - 16 octets)        |
+---------------------------------+----------------------------+
|     Port (2 octets, optional)   |
+---------------------------------+
```
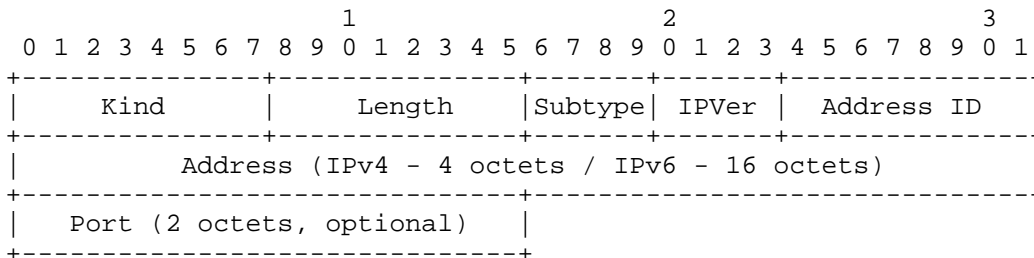
              Figure 12: Add Address (ADD_ADDR) Option

   Due to the proliferation of NATs, it is reasonably likely that one
   host may attempt to advertise private addresses [18]. It is not
   desirable to prohibit this, since there may be cases where both hosts
   have additional interfaces on the same private network, and a host
   MAY want to advertise such addresses. The MP_JOIN handshake to
   create a new subflow (Section 3.2) provides mechanisms to minimize
   security risks. The MP_JOIN message contains a 32-bit token that
   uniquely identifies the connection to the receiving host. If the
   token is unknown, the host will return with a RST. In the unlikely
   event that the token is known, subflow setup will continue, but the
   HMAC exchange must occur for authentication. This will fail, and
   will provide sufficient protection against two unconnected hosts
   accidentally setting up a new subflow upon the signal of a private
   address. Further security considerations around the issue of
   ADD_ADDR messages that accidentally misdirect, or maliciously direct,
   new MP_JOIN attempts are discussed in Section 5.

   Ideally, ADD_ADDR and REMOVE_ADDR options would be sent reliably, and
   in order, to the other end. This would ensure that this address
   management does not unnecessarily cause an outage in the connection
   when remove/add addresses are processed in reverse order, and also to
   ensure that all possible paths are used. Note, however, that losing
   reliability and ordering will not break the multipath connections, it
   will just reduce the opportunity to open multipath paths and to
   survive different patterns of path failures.

   Therefore, implementing reliability signals for these TCP options is
   not necessary. In order to minimize the impact of the loss of these
   options, however, it is RECOMMENDED that a sender should send these
   options on all available subflows. If these options need to be
   received in order, an implementation SHOULD only send one ADD_ADDR/
   REMOVE_ADDR option per RTT, to minimize the risk of misordering.

   A host can send an ADD_ADDR message with an already assigned Address
   ID, but the Address MUST be the same as previously assigned to this
   Address ID, and the Port MUST be different from one already in use

for this Address ID. If these conditions are not met, the receiver
SHOULD silently ignore the ADD_ADDR. A host wishing to replace an
existing Address ID MUST first remove the existing one
(Section 3.4.2).

A host that receives an ADD_ADDR but finds a connection set up to
that IP address and port number is unsuccessful SHOULD NOT perform
further connection attempts to this address/port combination for this
connection. A sender that wants to trigger a new incoming connection
attempt on a previously advertised address/port combination can
therefore refresh ADD_ADDR information by sending the option again.

During normal MPTCP operation, it is unlikely that there will be
sufficient TCP option space for ADD_ADDR to be included along with
those for data sequence numbering (Section 3.3.1). Therefore, it is
expected that an MPTCP implementation will send the ADD_ADDR option
on separate ACKs. As discussed earlier, however, an MPTCP
implementation MUST NOT treat duplicate ACKs with any MPTCP option,
with the exception of the DSS option, as indications of congestion
[12], and an MPTCP implementation SHOULD NOT send more than two
duplicate ACKs in a row for signaling purposes.

3.4.2. Remove Address

If, during the lifetime of an MPTCP connection, a previously
announced address becomes invalid (e.g., if the interface
disappears), the affected host SHOULD announce this so that the peer
can remove subflows related to this address.

This is achieved through the Remove Address (REMOVE_ADDR) option
(Figure 13), which will remove a previously added address (or list of
addresses) from a connection and terminate any subflows currently
using that address.

For security purposes, if a host receives a REMOVE_ADDR option, it
must ensure the affected path(s) are no longer in use before it
instigates closure. The receipt of REMOVE_ADDR SHOULD first trigger
the sending of a TCP keepalive [19] on the path, and if a response is
received the path SHOULD NOT be removed. Typical TCP validity tests
on the subflow (e.g., ensuring sequence and ACK numbers are correct)
MUST also be undertaken. An implementation can use indications of
these test failures as part of intrusion detection or error logging.

The sending and receipt (if no keepalive response was received) of
this message SHOULD trigger the sending of RSTs by both hosts on the
affected subflow(s) (if possible), as a courtesy to cleaning up
middlebox state, before cleaning up any local state.

   Address removal is undertaken by ID, so as to permit the use of NATs
   and other middleboxes that rewrite source addresses. If there is no
   address at the requested ID, the receiver will silently ignore the
   request.

   A subflow that is still functioning MUST be closed with a FIN
   exchange as in regular TCP, rather than using this option. For more
   information, see Section 3.3.3.

```
                         1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +---------------+---------------+-------+-------+---------------+
    |     Kind      |  Length = 3+n |Subtype|(resvd)|  Address ID   | ...
    +---------------+---------------+-------+-------+---------------+
                                     (followed by n-1 Address IDs, if required)
```
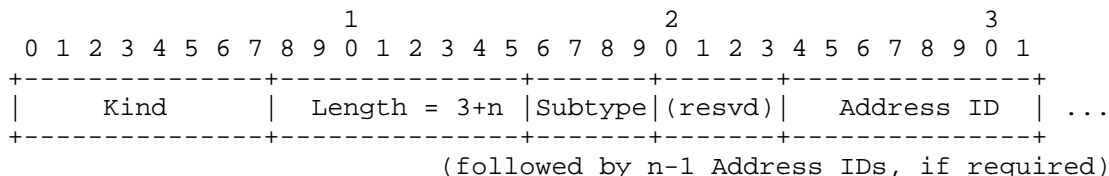
                Figure 13: Remove Address (REMOVE_ADDR) Option

3.5. Fast Close

   Regular TCP has the means of sending a reset (RST) signal to abruptly
   close a connection. With MPTCP, the RST only has the scope of the
   subflow and will only close the concerned subflow but not affect the
   remaining subflows. MPTCP's connection will stay alive at the data
   level, in order to permit break-before-make handover between
   subflows. It is therefore necessary to provide an MPTCP-level
   "reset" to allow the abrupt closure of the whole MPTCP connection,
   and this is the MP_FASTCLOSE option.

   MP_FASTCLOSE is used to indicate to the peer that the connection will
   be abruptly closed and no data will be accepted anymore. The reasons
   for triggering an MP_FASTCLOSE are implementation specific. Regular
   TCP does not allow sending a RST while the connection is in a
   synchronized state [1]. Nevertheless, implementations allow the
   sending of a RST in this state, if, for example, the operating system
   is running out of resources. In these cases, MPTCP should send the
   MP_FASTCLOSE. This option is illustrated in Figure 14.

```
                          1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +---------------+---------------+-------+-----------------------+
    |      Kind     |     Length    |Subtype|       (reserved)      |
    +---------------+---------------+-------+-----------------------+
    |                    Option Receiver's Key                      |
    |                           (64 bits)                           |
    |                                                               |
    +---------------------------------------------------------------+
```
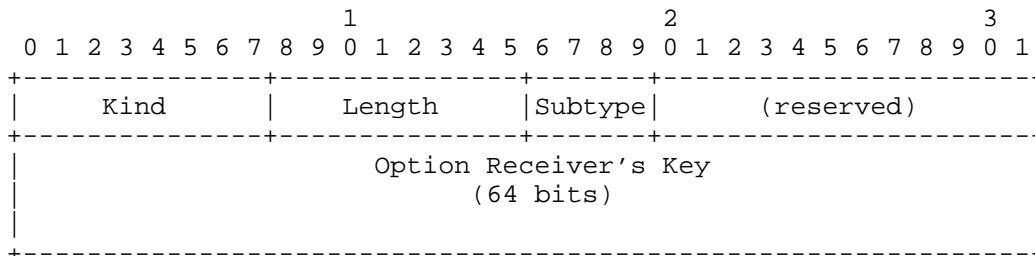
                Figure 14: Fast Close (MP_FASTCLOSE) Option

   If Host A wants to force the closure of an MPTCP connection, the
   MPTCP Fast Close procedure is as follows:

   o Host A sends an ACK containing the MP_FASTCLOSE option on one
      subflow, containing the key of Host B as declared in the initial
      connection handshake. On all the other subflows, Host A sends a
      regular TCP RST to close these subflows, and tears them down.
      Host A now enters FASTCLOSE_WAIT state.

   o Upon receipt of an MP_FASTCLOSE, containing the valid key, Host B
      answers on the same subflow with a TCP RST and tears down all
      subflows. Host B can now close the whole MPTCP connection (it
      transitions directly to CLOSED state).

   o As soon as Host A has received the TCP RST on the remaining
      subflow, it can close this subflow and tear down the whole
      connection (transition from FASTCLOSE_WAIT to CLOSED states). If
      Host A receives an MP_FASTCLOSE instead of a TCP RST, both hosts
      attempted fast closure simultaneously. Host A should reply with a
      TCP RST and tear down the connection.

   o If Host A does not receive a TCP RST in reply to its MP_FASTCLOSE
      after one retransmission timeout (RTO) (the RTO of the subflow
      where the MPTCP_RST has been sent), it SHOULD retransmit the
      MP_FASTCLOSE. The number of retransmissions SHOULD be limited to
      avoid this connection from being retained for a long time, but
      this limit is implementation specific. A RECOMMENDED number is 3.

3.6. Fallback

   Sometimes, middleboxes will exist on a path that could prevent the
   operation of MPTCP. MPTCP has been designed in order to cope with
   many middlebox modifications (see Section 6), but there are still
   some cases where a subflow could fail to operate within the MPTCP
   requirements. These cases are notably the following: the loss of TCP
   options on a path and the modification of payload data. If such an

event occurs, it is necessary to "fall back" to the previous, safe operation. This may be either falling back to regular TCP or removing a problematic subflow.

At the start of an MPTCP connection (i.e., the first subflow), it is important to ensure that the path is fully MPTCP capable and the necessary TCP options can reach each host. The handshake as described in Section 3.1 SHOULD fall back to regular TCP if either of the SYN messages do not have the MPTCP options: this is the same, and desired, behavior in the case where a host is not MPTCP capable, or the path does not support the MPTCP options. When attempting to join an existing MPTCP connection (Section 3.2), if a path is not MPTCP capable and the TCP options do not get through on the SYNs, the subflow will be closed according to the MP_JOIN logic.

There is, however, another corner case that should be addressed. That is one of MPTCP options getting through on the SYN, but not on regular packets. This can be resolved if the subflow is the first subflow, and thus all data in flight is contiguous, using the following rules.

A sender MUST include a DSS option with data sequence mapping in every segment until one of the sent segments has been acknowledged with a DSS option containing a Data ACK. Upon reception of the acknowledgment, the sender has the confirmation that the DSS option passes in both directions and may choose to send fewer DSS options than once per segment.

If, however, an ACK is received for data (not just for the SYN) without a DSS option containing a Data ACK, the sender determines the path is not MPTCP capable. In the case of this occurring on an additional subflow (i.e., one started with MP_JOIN), the host MUST close the subflow with a RST. In the case of the first subflow (i.e., that started with MP_CAPABLE), it MUST drop out of an MPTCP mode back to regular TCP. The sender will send one final data sequence mapping, with the Data-Level Length value of 0 indicating an infinite mapping (in case the path drops options in one direction only), and then revert to sending data on the single subflow without any MPTCP options.

Note that this rule essentially prohibits the sending of data on the third packet of an MP_CAPABLE or MP_JOIN handshake, since both that option and a DSS cannot fit in TCP option space. If the initiator is to send first, another segment must be sent that contains the data and DSS. Note also that an additional subflow cannot be used until the initial path has been verified as MPTCP capable.

These rules should cover all cases where such a failure could happen:
whether it's on the forward or reverse path and whether the server or
the client first sends data. If lost options on data packets occur
on any other subflow apart from the initial subflow, it should be
treated as a standard path failure. The data would not be DATA_ACKed
(since there is no mapping for the data), and the subflow can be
closed with a RST.

The case described above is a specialized case of fallback, for when
the lack of MPTCP support is detected before any data is acknowledged
at the connection level on a subflow. More generally, fallback
(either closing a subflow, or to regular TCP) can become necessary at
any point during a connection if a non-MPTCP-aware middlebox changes
the data stream.

As described in Section 3.3, each portion of data for which there is
a mapping is protected by a checksum. This mechanism is used to
detect if middleboxes have made any adjustments to the payload
(added, removed, or changed data). A checksum will fail if the data
has been changed in any way. This will also detect if the length of
data on the subflow is increased or decreased, and this means the
data sequence mapping is no longer valid. The sender no longer knows
what subflow-level sequence number the receiver is genuinely
operating at (the middlebox will be faking ACKs in return), and it
cannot signal any further mappings. Furthermore, in addition to the
possibility of payload modifications that are valid at the
application layer, there is the possibility that false positives
could be hit across MPTCP segment boundaries, corrupting the data.
Therefore, all data from the start of the segment that failed the
checksum onwards is not trustworthy.

When multiple subflows are in use, the data in flight on a subflow
will likely involve data that is not contiguously part of the
connection-level stream, since segments will be spread across the
multiple subflows. Due to the problems identified above, it is not
possible to determine what the adjustment has done to the data
(notably, any changes to the subflow sequence numbering). Therefore,
it is not possible to recover the subflow, and the affected subflow
must be immediately closed with a RST, featuring an MP_FAIL option
(Figure 15), which defines the data sequence number at the start of
the segment (defined by the data sequence mapping) that had the
checksum failure. Note that the MP_FAIL option requires the use of
the full 64-bit sequence number, even if 32-bit sequence numbers are
normally in use in the DSS signals on the path.

```
                         1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +---------------+---------------+-------+----------------------+
    |      Kind     |   Length=12   |Subtype|      (reserved)      |
    +---------------+---------------+-------+----------------------+
    |                                                              |
    |                Data Sequence Number (8 octets)               |
    |                                                              |
    +--------------------------------------------------------------+
```
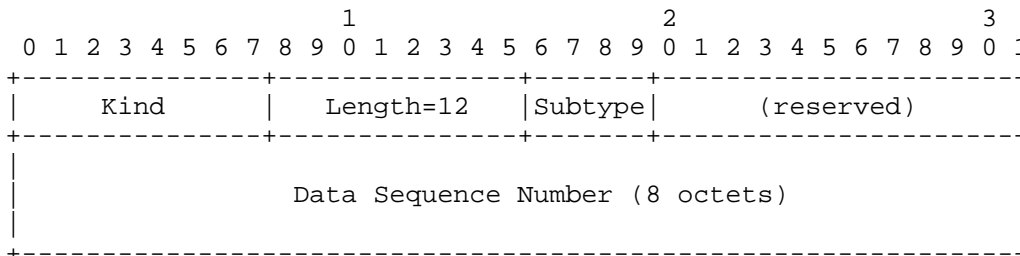
                  Figure 15: Fallback (MP_FAIL) Option

The receiver MUST discard all data following the data sequence number
specified. Failed data MUST NOT be DATA_ACKed and so will be
retransmitted on other subflows (Section 3.3.6).

A special case is when there is a single subflow and it fails with a
checksum error. If it is known that all unacknowledged data in
flight is contiguous (which will usually be the case with a single
subflow), an infinite mapping can be applied to the subflow without
the need to close it first, and essentially turn off all further
MPTCP signaling. In this case, if a receiver identifies a checksum
failure when there is only one path, it will send back an MP_FAIL
option on the subflow-level ACK, referring to the data-level sequence
number of the start of the segment on which the checksum error was
detected. The sender will receive this, and if all unacknowledged
data in flight is contiguous, will signal an infinite mapping. This
infinite mapping will be a DSS option (Section 3.3) on the first new
packet, containing a data sequence mapping that acts retroactively,
referring to the start of the subflow sequence number of the last
segment that was known to be delivered intact. From that point
onwards, data can be altered by a middlebox without affecting MPTCP,
as the data stream is equivalent to a regular, legacy TCP session.

In the rare case that the data is not contiguous (which could happen
when there is only one subflow but it is retransmitting data from a
subflow that has recently been uncleanly closed), the receiver MUST
close the subflow with a RST with MP_FAIL. The receiver MUST discard
all data that follows the data sequence number specified. The sender
MAY attempt to create a new subflow belonging to the same connection,
and, if it chooses to do so, SHOULD place the single subflow
immediately in single-path mode by setting an infinite data sequence
mapping. This mapping will begin from the data-level sequence number
that was declared in the MP_FAIL.

After a sender signals an infinite mapping, it MUST only use subflow
ACKs to clear its send buffer. This is because Data ACKs may become
misaligned with the subflow ACKs when middleboxes insert or delete
data. The receive SHOULD stop generating Data ACKs after it receives
an infinite mapping.

When a connection has fallen back, only one subflow can send data;
otherwise, the receiver would not know how to reorder the data. In
practice, this means that all MPTCP subflows will have to be
terminated except one. Once MPTCP falls back to regular TCP, it MUST
NOT revert to MPTCP later in the connection.

It should be emphasized that we are not attempting to prevent the use
of middleboxes that want to adjust the payload. An MPTCP-aware
middlebox could provide such functionality by also rewriting
checksums.

3.7. Error Handling

In addition to the fallback mechanism as described above, the
standard classes of TCP errors may need to be handled in an MPTCP-
specific way. Note that changing semantics -- such as the relevance
of a RST -- are covered in Section 4. Where possible, we do not want
to deviate from regular TCP behavior.

The following list covers possible errors and the appropriate MPTCP
behavior:

o Unknown token in MP_JOIN (or HMAC failure in MP_JOIN ACK, or
   missing MP_JOIN in SYN/ACK response): send RST (analogous to TCP's
   behavior on an unknown port)

o DSN out of window (during normal operation): drop the data, do not
   send Data ACKs

o Remove request for unknown address ID: silently ignore

3.8. Heuristics

There are a number of heuristics that are needed for performance or
deployment but that are not required for protocol correctness. In
this section, we detail such heuristics. Note that discussion of
buffering and certain sender and receiver window behaviors are
presented in Sections 3.3.4 and 3.3.5, as well as retransmission in
Section 3.3.6.

3.8.1. Port Usage

   Under typical operation, an MPTCP implementation SHOULD use the same
   ports as already in use. In other words, the destination port of a
   SYN containing an MP_JOIN option SHOULD be the same as the remote
   port of the first subflow in the connection. The local port for such
   SYNs SHOULD also be the same as for the first subflow (and as such,
   an implementation SHOULD reserve ephemeral ports across all local IP
   addresses), although there may be cases where this is infeasible.
   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.

   There may also be cases, however, where the passive opener wishes to
   signal to the other host that a specific port should be used, and
   this facility is provided in the Add Address option as documented in
   Section 3.4.1. It is therefore feasible to allow multiple subflows
   between the same two addresses but using different port pairs, and
   such a facility could be used to allow load balancing within the
   network based on 5-tuples (e.g., some ECMP implementations [7]).

3.8.2. Delayed Subflow Start

   Many TCP connections are short-lived and consist only of a few
   segments, and so the overheads of using MPTCP outweigh any benefits.
   A heuristic is required, therefore, to decide when to start using
   additional subflows in an MPTCP connection. We expect that
   experience gathered from deployments will provide further guidance on
   this, and will be affected by particular application characteristics
   (which are likely to change over time). However, a suggested
   general-purpose heuristic that an implementation MAY choose to employ
   is as follows. Results from experimental deployments are needed in
   order to verify the correctness of this proposal.

   If a host has data buffered for its peer (which implies that the
   application has received a request for data), the host opens one
   subflow for each initial window's worth of data that is buffered.

   Consideration should also be given to limiting the rate of adding new
   subflows, as well as limiting the total number of subflows open for a
   particular connection. A host may choose to vary these values based
   on its load or knowledge of traffic and path characteristics.

   Note that this heuristic alone is probably insufficient. Traffic for
   many common applications, such as downloads, is highly asymmetric and
   the host that is multihomed may well be the client that will never

fill its buffers, and thus never use MPTCP. Advanced APIs that allow
an application to signal its traffic requirements would aid in these
decisions.

An additional time-based heuristic could be applied, opening
additional subflows after a given period of time has passed. This
would alleviate the above issue, and also provide resilience for low-
bandwidth but long-lived applications.

This section has shown some of the considerations that an implementer
should give when developing MPTCP heuristics, but is not intended to
be prescriptive.

3.8.3. Failure Handling

Requirements for MPTCP's handling of unexpected signals have been
given in Section 3.7. There are other failure cases, however, where
a hosts can choose appropriate behavior.

For example, Section 3.1 suggests that a host SHOULD fall back to
trying regular TCP SYNs after one or more failures of MPTCP SYNs for
a connection. A host may keep a system-wide cache of such
information, so that it can back off from using MPTCP, firstly for
that particular destination host, and eventually on a whole
interface, if MPTCP connections continue failing.

Another failure could occur when the MP_JOIN handshake fails.
Section 3.7 specifies that an incorrect handshake MUST lead to the
subflow being closed with a RST. A host operating an active
intrusion detection system may choose to start blocking MP_JOIN
packets from the source host if multiple failed MP_JOIN attempts are
seen. From the connection initiator's point of view, if an MP_JOIN
fails, it SHOULD NOT attempt to connect to the same IP address and
port during the lifetime of the connection, unless the other host
refreshes the information with another ADD_ADDR option. Note that
the ADD_ADDR option is informational only, and does not guarantee the
other host will attempt a connection.

In addition, an implementation may learn, over a number of
connections, that certain interfaces or destination addresses
consistently fail and may default to not trying to use MPTCP for
these. Behavior could also be learned for particularly badly
performing subflows or subflows that regularly fail during use, in
order to temporarily choose not to use these paths.

4. Semantic Issues

   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 1 octet of sequence space. There is an explicit mapping of
      data sequence space to subflow sequence space, which is signaled
      through TCP options in data packets.

   ACK: The ACK field in the TCP header acknowledges only the subflow
      sequence number, not the data-level sequence space.
      Implementations SHOULD NOT attempt to infer a data-level
      acknowledgment from the subflow ACKs. This separates subflow- and
      connection-level processing at an end host.

   Duplicate ACK: A duplicate ACK that includes any MPTCP signaling
      (with the exception of the DSS option) MUST NOT be treated as a
      signal of congestion. To limit the chances of non-MPTCP-aware
      entities mistakenly interpreting duplicate ACKs as a signal of
      congestion, MPTCP SHOULD NOT send more than two duplicate ACKs
      containing (non-DSS) MPTCP signals in a row.

   Receive Window: The receive window in the TCP header indicates the
      amount of free buffer space for the whole data-level connection
      (as opposed to for this subflow) that is available at the
      receiver. This is the same semantics as regular TCP, but to
      maintain these semantics the receive window must be interpreted at
      the sender as relative to the sequence number given in the
      DATA_ACK rather than the subflow ACK in the TCP header. In this
      way, the original flow control role is preserved. Note that some
      middleboxes may change the receive window, and so a host SHOULD
      use the maximum value of those recently seen on the constituent
      subflows for the connection-level receive window, and also needs
      to maintain a subflow-level window for subflow-level processing.

   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. The MP_FASTCLOSE option
      provides the fast close functionality of a RST at the MPTCP
      connection level.

   Address List: Address list management (i.e., knowledge of the local
      and remote hosts' lists of available IP addresses) 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
      (either explicitly through an Add Address message, or implicitly
      through a Join) has no implication for other connections between
      the same pair of hosts.

   5-tuple: The 5-tuple (protocol, local address, local port, remote
      address, remote port) presented by kernel APIs 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. This decision, and other related API issues,
      are discussed in more detail in [6].

5. Security Considerations

   As identified in [9], the addition of multipath capability to TCP
   will bring with it a number of new classes of threat. In order to
   prevent these, [2] presents a set of requirements for a security
   solution for MPTCP. The fundamental goal is for the security of
   MPTCP to be "no worse" than regular TCP today, and the key security
   requirements are:

   o Provide a mechanism to confirm that the parties in a subflow
      handshake are the same as in the original connection setup.

   o Provide verification that the peer can receive traffic at a new
      address before using it as part of a connection.

   o Provide replay protection, i.e., ensure that a request to add/
      remove a subflow is 'fresh'.

   In order to achieve these goals, MPTCP includes a hash-based
   handshake algorithm documented in Sections 3.1 and 3.2.

   The security of the MPTCP connection hangs on the use of keys that
   are shared once at the start of the first subflow, and are never sent
   again over the network (unless used in the fast close mechanism,
   Section 3.5). To ease demultiplexing while not giving away any
   cryptographic material, future subflows use a truncated cryptographic
   hash of this key as the connection identification "token". The keys
   are concatenated and used as keys for creating Hash-based Message
   Authentication Codes (HMACs) used on subflow setup, in order to
   verify that the parties in the handshake are the same as in the
   original connection setup. It also provides verification that the
   peer can receive traffic at this new address. Replay attacks would

still be possible when only keys are used; therefore, the handshakes
use single-use random numbers (nonces) at both ends -- this ensures
the HMAC will never be the same on two handshakes. Guidance on
generating random numbers suitable for use as keys is given in [14]
and discussed in Section 3.1.

The use of crypto capability bits in the initial connection handshake
to negotiate use of a particular algorithm allows the deployment of
additional crypto mechanisms in the future. Note that this would be
susceptible to bid-down attacks only if the attacker was on-path (and
thus would be able to modify the data anyway). The security
mechanism presented in this document should therefore protect against
all forms of flooding and hijacking attacks discussed in [9].

During normal operation, regular TCP protection mechanisms (such as
ensuring sequence numbers are in-window) will provide the same level
of protection against attacks on individual TCP subflows as exists
for regular TCP today. Implementations will introduce additional
buffers compared to regular TCP, to reassemble data at the connection
level. The application of window sizing will minimize the risk of
denial-of-service attacks consuming resources.

As discussed in Section 3.4.1, a host may advertise its private
addresses, but these might point to different hosts in the receiver's
network. The MP_JOIN handshake (Section 3.2) will ensure that this
does not succeed in setting up a subflow to the incorrect host.
However, it could still create unwanted TCP handshake traffic. This
feature of MPTCP could be a target for denial-of-service exploits,
with malicious participants in MPTCP connections encouraging the
recipient to target other hosts in the network. Therefore,
implementations should consider heuristics (Section 3.8) at both the
sender and receiver to reduce the impact of this.

A small security risk could theoretically exist with key reuse, but
in order to accomplish a replay attack, both the sender and receiver
keys, and the sender and receiver random numbers, in the MP_JOIN
handshake (Section 3.2) would have to match.

Whilst this specification defines a "medium" security solution,
meeting the criteria specified at the start of this section and the
threat analysis ([9]), since attacks only ever get worse, it is
likely that a future Standards Track version of MPTCP would need to
be able to support stronger security. There are several ways the
security of MPTCP could potentially be improved; some of these would
be compatible with MPTCP as defined in this document, whilst others
may not be. For now, the best approach is to get experience with the
current approach, establish what might work, and check that the
threat analysis is still accurate.

   Possible ways of improving MPTCP security could include:

   o defining a new MPCTP cryptographic algorithm, as negotiated in
      MP_CAPABLE. A sub-case could be to include an additional
      deployment assumption, such as stateful servers, in order to allow
      a more powerful algorithm to be used.

   o defining how to secure data transfer with MPTCP, whilst not
      changing the signaling part of the protocol.

   o defining security that requires more option space, perhaps in
      conjunction with a "long options" proposal for extending the TCP
      options space (such as those surveyed in [20]), or perhaps
      building on the current approach with a second stage of MPTCP-
      option-based security.

   o revisiting the working group's decision to exclusively use TCP
      options for MPTCP signaling, and instead look at also making use
      of the TCP payloads.

   MPTCP has been designed with several methods available to indicate a
   new security mechanism, including:

   o available flags in MP_CAPABLE (Figure 4);

   o available subtypes in the MPTCP option (Figure 3);

   o the version field in MP_CAPABLE (Figure 4);

6. Interactions with Middleboxes

   Multipath TCP was designed to be deployable in the present world.
   Its design takes into account "reasonable" existing middlebox
   behavior. 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 accommodate the different middleboxes.

   A primary concern is our use of a new TCP option. Middleboxes should
   forward packets with unknown 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 uses a single new TCP option "Kind", and all message types are
defined by "subtype" values (see Section 8). This should reduce the
chances of only some types of MPTCP options being passed, and instead
the key differing characteristics are different paths, and the
presence of the SYN flag.

MPTCP SYN packets on the first subflow of a connection contain the
MP_CAPABLE option (Section 3.1). If this is dropped, MPTCP SHOULD
fall back to regular TCP. If packets with the MP_JOIN option
(Section 3.2) are dropped, the paths will simply not be used.

If a middlebox strips options but otherwise passes the packets
unchanged, MPTCP will behave safely. If an MP_CAPABLE option is
dropped on either the outgoing or the return path, the initiating
host can fall back to regular TCP, as illustrated in Figure 16 and
discussed in Section 3.1.

Subflow SYNs contain the MP_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 MP_JOIN option and its
token. Either way, the subflow setup fails, but otherwise does not
affect the MPTCP connection as a whole.

```
      Host A                               Host B
        |                Middlebox M          |
        |                    |                |
        |   SYN(MP_CAPABLE)  |      SYN        |
        |------------------|---------------->|
        |                  SYN/ACK             |
        |<------------------------------------|
     a) MP_CAPABLE option stripped on outgoing path


       Host A                               Host B
         |              SYN(MP_CAPABLE)         |
         |----------------------------------->|
         |              Middlebox M            |
         |                  |                  |
         |    SYN/ACK       |SYN/ACK(MP_CAPABLE)|
         |<---------------|------------------|
       b) MP_CAPABLE option stripped on return path
```
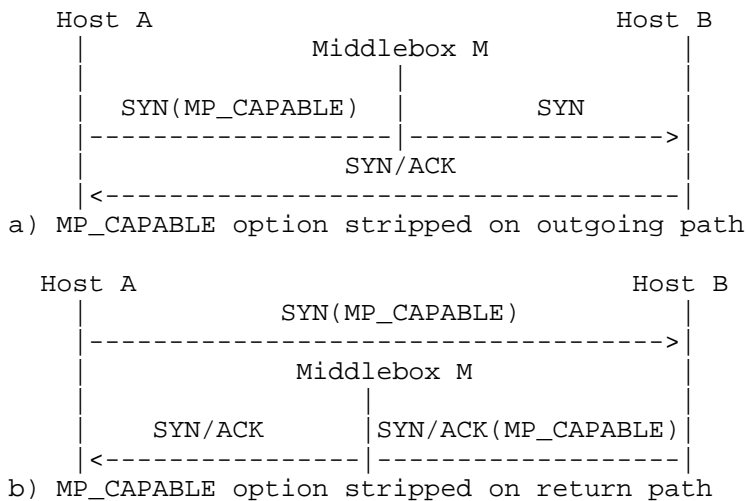
        Figure 16: Connection Setup with Middleboxes that
                Strip Options from Packets

We now examine data flow with MPTCP, assuming the flow is correctly
set up, 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.

The case when options get stripped on data packets has been discussed
in the Fallback section. If a fraction of options are stripped,
behavior is not deterministic. If some data sequence mappings are
lost, the connection can continue so long as mappings exist for the
subflow-level data (e.g., if multiple maps have been sent that
reinforce each other). If some subflow-level space is left unmapped,
however, the subflow is treated as broken and is closed, through the
process described in Section 3.6. MPTCP should survive with a loss
of some Data ACKs, 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 behavior,
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:

o NATs [21] (Network Address (and Port) Translators) change the
   source address (and often source port) of packets. This means
   that a host will not know its public-facing address for signaling
   in MPTCP. Therefore, MPTCP permits implicit address addition via
   the MP_JOIN option, and the handshake mechanism ensures that
   connection attempts to private addresses [18] do not cause
   problems. Explicit address removal is undertaken by an Address ID
   to allow no knowledge of the source address.

o Performance Enhancing Proxies (PEPs) [22] might proactively ACK
   data to increase performance. MPTCP, however, relies on accurate
   congestion control signals from the end host, and non-MPTCP-aware
   PEPs will not be able to provide such signals. MPTCP will,
   therefore, fall back to single-path TCP, or close the problematic
   subflow (see Section 3.6).

o Traffic Normalizers [23] may not allow holes in sequence numbers,
   and may 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. In the event of a
   retransmission, the same data will be retransmitted on the
   original TCP subflow even if it is additionally retransmitted at
   the connection level on a different subflow.

   o Firewalls [24] might perform initial sequence number randomization
      on TCP connections. MPTCP uses relative sequence numbers in data
      sequence mapping to cope with this. Like NATs, firewalls will not
      permit many incoming connections, so MPTCP supports address
      signaling (ADD_ADDR) so that a multiaddressed host can invite its
      peer behind the firewall/NAT to connect out to its additional
      interface.

   o Intrusion Detection Systems look out for traffic patterns and
      content that could threaten a network. Multipath will mean that
      such data is potentially spread, so it is more difficult for an
      IDS to analyze the whole traffic, and potentially increases the
      risk of false positives. However, for an MPTCP-aware IDS, tokens
      can be read by such systems to correlate multiple subflows and
      reassemble for analysis.

   o Application-level middleboxes such as content-aware firewalls may
      alter the payload within a subflow, such as rewriting URIs in HTTP
      traffic. MPTCP will detect these using the checksum and close the
      affected subflow(s), if there are other subflows that can be used.
      If all subflows are affected, multipath will fall back to TCP,
      allowing such middleboxes to change the payload. MPTCP-aware
      middleboxes should be able to adjust the payload and MPTCP
      metadata in order not to break the connection.

   In addition, all classes of middleboxes may affect TCP traffic in the
   following ways:

   o TCP options may be removed, or packets with unknown options
      dropped, by many classes of middleboxes. It is intended that the
      initial SYN exchange, with a TCP option, will be sufficient to
      identify the path capabilities. If such a packet does not get
      through, MPTCP will end up falling back to regular TCP.

   o Segmentation/Coalescing (e.g., TCP segmentation offloading) might
      copy options between packets and might strip some options.
      MPTCP's data sequence mapping includes the relative 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.

   o The receive window may be shrunk by some middleboxes at the
      subflow level. MPTCP will use the maximum window at data level,
      but will also obey subflow-specific windows.

7. Acknowledgments

   The authors were originally supported by Trilogy
   (http://www.trilogy-project.org), a research project (ICT-216372)
   partially funded by the European Community under its Seventh
   Framework Program.

   Alan Ford was originally supported by Roke Manor Research.

   The authors gratefully acknowledge significant input into this
   document from Sebastien Barre, Christoph Paasch, and Andrew McDonald.

   The authors also wish to acknowledge reviews and contributions from
   Iljitsch van Beijnum, Lars Eggert, Marcelo Bagnulo, Robert Hancock,
   Pasi Sarolahti, Toby Moncaster, Philip Eardley, Sergio Lembo,
   Lawrence Conroy, Yoshifumi Nishida, Bob Briscoe, Stein Gjessing,
   Andrew McGregor, Georg Hampel, Anumita Biswas, Wes Eddy, Alexey
   Melnikov, Francis Dupont, Adrian Farrel, Barry Leiba, Robert Sparks,
   Sean Turner, Stephen Farrell, and Martin Stiemerling.

8. IANA Considerations

   This document defines a new TCP option for MPTCP, assigned a value of
   30 (decimal) from the TCP option space. This value is the value of
   "Kind" as seen in all MPTCP options in this document. This value is
   defined as:

```
           +------+--------+----------------------+-----------+
           | Kind | Length |       Meaning        | Reference |
           +------+--------+----------------------+-----------+
           |  30  |   N    | Multipath TCP (MPTCP)|  RFC 6824 |
           +------+--------+----------------------+-----------+
```

                    Table 1: TCP Option Kind Numbers

   This document also defines a 4-bit subtype field, for which IANA has
   created and will maintain a new sub-registry entitled "MPTCP Option
   Subtypes" under the "Transmission Control Protocol (TCP) Parameters"
   registry. Initial values for the MPTCP option subtype registry are
   given below; future assignments are to be defined by Standards Action
   as defined by [25]. Assignments consist of the MPTCP subtype's
   symbolic name and its associated value, as per the following table.

```
+-------+-------------+---------------------------+---------------+
| Value |   Symbol    |           Name            |   Reference   |
+-------+-------------+---------------------------+---------------+
|  0x0  | MP_CAPABLE  |      Multipath Capable    | Section 3.1   |
|  0x1  | MP_JOIN     |      Join Connection      | Section 3.2   |
|  0x2  |    DSS      | Data Sequence Signal (Data| Section 3.3   |
|       |             |    ACK and data sequence  |               |
|       |             |         mapping)          |               |
|  0x3  | ADD_ADDR    |       Add Address         | Section 3.4.1 |
|  0x4  | REMOVE_ADDR |      Remove Address       | Section 3.4.2 |
|  0x5  | MP_PRIO     | Change Subflow Priority   | Section 3.3.8 |
|  0x6  | MP_FAIL     |        Fallback           | Section 3.6   |
|  0x7  | MP_FASTCLOSE|        Fast Close         | Section 3.5   |
+-------+-------------+---------------------------+---------------+
```

                    Table 2: MPTCP Option Subtypes

Values 0x8 through 0xe are currently unassigned. The value 0xf is
reserved for Private Use within controlled testbeds.

IANA has created another sub-registry, "MPTCP Handshake Algorithms"
under the "Transmission Control Protocol (TCP) Parameters" registry,
based on the flags in MP_CAPABLE (Section 3.1). The flags consist of
8 bits, labeled "A" through "H", and this document assigns the bits
as follows:

```
+----------+------------------+----------------------+
| Flag Bit |     Meaning      |      Reference       |
+----------+------------------+----------------------+
|    A     | Checksum required| RFC 6824, Section 3.1|
|    B     |  Extensibility   | RFC 6824, Section 3.1|
|   C-G    |    Unassigned    |                      |
|    H     |   HMAC-SHA1      | RFC 6824, Section 3.2|
+----------+------------------+----------------------+
```

                 Table 3: MPTCP Handshake Algorithms

Note that the meanings of bits C through H can be dependent upon bit
B, depending on how Extensibility is defined in future
specifications; see Section 3.1 for more information.

Future assignments in this registry are also to be defined by
Standards Action as defined by [25]. Assignments consist of the
value of the flags, a symbolic name for the algorithm, and a
reference to its specification.

9. References

9.1. Normative References

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

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

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

   [4]   National Institute of Science and Technology, "Secure Hash
          Standard", Federal Information Processing Standard
          (FIPS) 180-3, October 2008, <http://csrc.nist.gov/publications/
          fips/fips180-3/fips180-3_final.pdf>.

9.2. Informative References

   [5]   Raiciu, C., Handley, M., and D. Wischik, "Coupled Congestion
          Control for Multipath Transport Protocols", RFC 6356,
          October 2011.

   [6]   Scharf, M. and A. Ford, "MPTCP Application Interface
          Considerations", Work in Progress, October 2012.

   [7]   Hopps, C., "Analysis of an Equal-Cost Multi-Path Algorithm",
          RFC 2992, November 2000.

   [8]   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",
          Usenix Symposium on Networked Systems Design and
          Implementation 012, 2012, <https://www.usenix.org/conference/
          nsdi12/how-hard-can-it-be-designing-and-implementing-
          deployable-multipath-tcp>.

   [9]   Bagnulo, M., "Threat Analysis for TCP Extensions for Multipath
          Operation with Multiple Addresses", RFC 6181, March 2011.

   [10]   Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing
          for Message Authentication", RFC 2104, February 1997.

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

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

   [13] Gont, F., "Survey of Security Hardening Methods for
         Transmission Control Protocol (TCP) Implementations", Work
         in Progress, March 2012.

   [14] Eastlake, D., Schiller, J., and S. Crocker, "Randomness
         Requirements for Security", BCP 106, RFC 4086, June 2005.

   [15] Eastlake, D. and T. Hansen, "US Secure Hash Algorithms (SHA and
         SHA-based HMAC and HKDF)", RFC 6234, May 2011.

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

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

   [18] Rekhter, Y., Moskowitz, R., Karrenberg, D., Groot, G., and E.
         Lear, "Address Allocation for Private Internets", BCP 5,
         RFC 1918, February 1996.

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

   [20] Ramaiah, A., "TCP option space extension", Work in Progress,
         March 2012.

   [21] Srisuresh, P. and K. Egevang, "Traditional IP Network Address
         Translator (Traditional NAT)", RFC 3022, January 2001.

   [22] Border, J., Kojo, M., Griner, J., Montenegro, G., and Z.
         Shelby, "Performance Enhancing Proxies Intended to Mitigate
         Link-Related Degradations", RFC 3135, June 2001.

   [23] Handley, M., Paxson, V., and C. Kreibich, "Network Intrusion
         Detection: Evasion, Traffic Normalization, and End-to-End
         Protocol Semantics", Usenix Security 2001, 2001,
         <http://www.usenix.org/events/sec01/full_papers/
         handley/handley.pdf>.

   [24] Freed, N., "Behavior of and Requirements for Internet
         Firewalls", RFC 2979, October 2000.

   [25] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA
         Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

Appendix A. Notes on Use of TCP Options

   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
   SYN, data, and pure ACK packets, and found that there is enough room
   to fit all the options we propose using in this document.

   SYN packets typically include Maximum Segment Size (MSS) (4 bytes),
   window scale (3 bytes), SACK permitted (2 bytes), and timestamp (10
   bytes) options. Together these sum to 19 bytes. Some operating
   systems appear to pad each option up to a word boundary, thus using
   24 bytes (a brief survey suggests Windows XP and Mac OS X do this,
   whereas Linux does not). Optimistically, therefore, we have 21 bytes
   spare, or 16 if it has to be word-aligned. In either case, however,
   the SYN versions of Multipath Capable (12 bytes) and Join (12 or 16
   bytes) options will fit in this remaining space.

   TCP data packets typically carry timestamp options in every packet,
   taking 10 bytes (or 12 with padding). That leaves 30 bytes (or 28,
   if word-aligned). The Data Sequence Signal (DSS) option varies in
   length depending on whether the data sequence mapping and DATA_ACK
   are included, and whether the sequence numbers in use are 4 or 8
   octets. The maximum size of the DSS option is 28 bytes, so even that
   will fit in the available space. But unless a connection is both
   bidirectional and high-bandwidth, it is unlikely that all that option
   space will be required on each DSS option.

   Within the DSS option, it is not necessary to include the data
   sequence mapping and DATA_ACK in each packet, and in many cases it
   may be possible to alternate their presence (so long as the mapping
   covers the data being sent in the following packet). It would also
   be possible to alternate between 4- and 8-byte sequence numbers in
   each option.

   On subflow and connection setup, an MPTCP option is also set on the
   third packet (an ACK). These are 20 bytes (for Multipath Capable)
   and 24 bytes (for Join), both of which will fit in the available
   option space.

   Pure ACKs in TCP typically contain only timestamps (10 bytes). Here,
   Multipath TCP typically needs to encode only the DATA_ACK (maximum of
   12 bytes). Occasionally, ACKs will contain SACK information.
   Depending on the number of lost packets, SACK may utilize the entire

   option space. If a DATA_ACK had to be included, then it is probably
   necessary to reduce the number of SACK blocks to accommodate the
   DATA_ACK. However, the presence of the DATA_ACK is unlikely to be
   necessary in a case where SACK is in use, since until at least some
   of the SACK blocks have been retransmitted, the cumulative data-level
   ACK will not be moving forward (or if it does, due to retransmissions
   on another path, then that path can also be used to transmit the new
   DATA_ACK).

   The ADD_ADDR option can be between 8 and 22 bytes, depending on
   whether IPv4 or IPv6 is used, and whether or not the port number is
   present. It is unlikely that such signaling would fit in a data
   packet (although if there is space, it is fine to include it). It is
   recommended to use duplicate ACKs with no other payload or options in
   order to transmit these rare signals. Note this is the reason for
   mandating that duplicate ACKs with MPTCP options are not taken as a
   signal of congestion.

   Finally, there are issues with reliable delivery of options. 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_ADDR/REMOVE_ADDR options. Here, it is
   recommended to send these options redundantly (whether on multiple
   paths or on the same path on a number of ACKs -- but interspersed
   with data in order to avoid interpretation as congestion). The cases
   where options are stripped by middleboxes are discussed in Section 6.

Appendix B. Control Blocks

   Conceptually, an MPTCP connection can be represented as an MPTCP
   control block that contains several variables that track the progress
   and the state of the MPTCP connection and a set of linked TCP control
   blocks that correspond to the subflows that have been established.

   RFC 793 [1] specifies several state variables. Whenever possible, we
   reuse the same terminology as RFC 793 to describe the state variables
   that are maintained by MPTCP.

B.1. MPTCP Control Block

   The MPTCP control block contains the following variable per
   connection.

B.1.1. Authentication and Metadata

   Local.Token (32 bits): This is the token chosen by the local host on
      this MPTCP connection. The token MUST be unique among all
      established MPTCP connections, generated from the local key.

   Local.Key (64 bits): This is the key sent by the local host on this
      MPTCP connection.

   Remote.Token (32 bits): This is the token chosen by the remote host
      on this MPTCP connection, generated from the remote key.

   Remote.Key (64 bits): This is the key chosen by the remote host on
      this MPTCP connection

   MPTCP.Checksum (flag): This flag is set to true if at least one of
      the hosts has set the C bit in the MP_CAPABLE options exchanged
      during connection establishment, and is set to false otherwise.
      If this flag is set, the checksum must be computed in all DSS
      options.

B.1.2. Sending Side

   SND.UNA (64 bits): This is the data sequence number of the next byte
      to be acknowledged, at the MPTCP connection level. This variable
      is updated upon reception of a DSS option containing a DATA_ACK.

   SND.NXT (64 bits): This is the data sequence number of the next byte
      to be sent. SND.NXT is used to determine the value of the DSN in
      the DSS option.

   SND.WND (32 bits with RFC 1323, 16 bits otherwise): This is the
      sending window. MPTCP maintains the sending window at the MPTCP
      connection level and the same window is shared by all subflows.
      All subflows use the MPTCP connection level SND.WND to compute the
      SEQ.WND value that is sent in each transmitted segment.

B.1.3. Receiving Side

   RCV.NXT (64 bits): This is the data sequence number of the next byte
      that is expected on the MPTCP connection. This state variable is
      modified upon reception of in-order data. The value of RCV.NXT is
      used to specify the DATA_ACK that is sent in the DSS option on all
      subflows.

   RCV.WND (32 bits with RFC 1323, 16 bits otherwise): This is the
      connection-level receive window, which is the maximum of the
      RCV.WND on all the subflows.

B.2. TCP Control Blocks

   The MPTCP control block also contains a list of the TCP control
   blocks that are associated to the MPTCP connection.

   Note that the TCP control block on the TCP subflows does not contain
   the RCV.WND and SND.WND state variables as these are maintained at
   the MPTCP connection level and not at the subflow level.

   Inside each TCP control block, the following state variables are
   defined.

B.2.1. Sending Side

   SND.UNA (32 bits): This is the sequence number of the next byte to
      be acknowledged on the subflow. This variable is updated upon
      reception of each TCP acknowledgment on the subflow.

   SND.NXT (32 bits): This is the sequence number of the next byte to
      be sent on the subflow. SND.NXT is used to set the value of
      SEG.SEQ upon transmission of the next segment.

B.2.2. Receiving Side

   RCV.NXT (32 bits): This is the sequence number of the next byte that
      is expected on the subflow. This state variable is modified upon
      reception of in-order segments. The value of RCV.NXT is copied to
      the SEG.ACK field of the next segments transmitted on the subflow.

   RCV.WND (32 bits with RFC 1323, 16 bits otherwise): This is the
      subflow-level receive window that is updated with the window field
      from the segments received on this subflow.

Appendix C. Finite State Machine

   The diagram in Figure 17 shows the Finite State Machine for
   connection-level closure. This illustrates how the DATA_FIN
   connection-level signal (indicated as the DFIN flag on a DATA_ACK)
   interacts with subflow-level FINs, and permits "break-before-make"
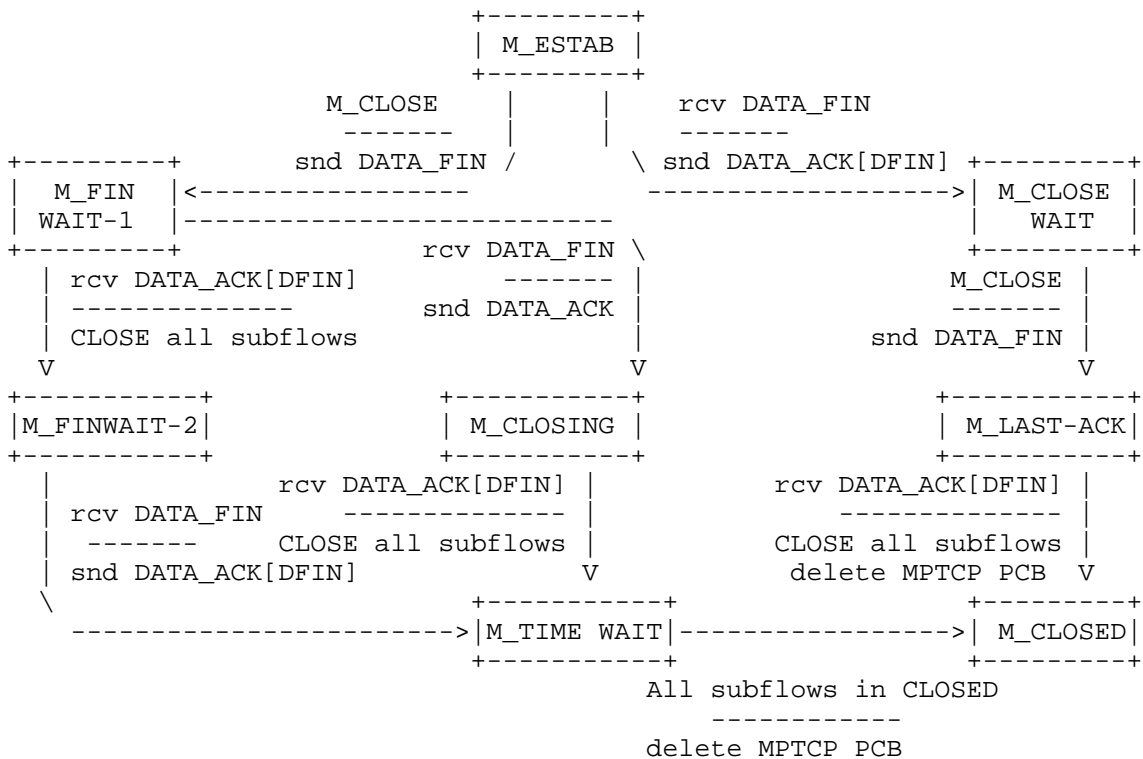   handover between subflows.

```
                                +---------+
                                | M_ESTAB |
                                +---------+
                    M_CLOSE      |   |   rcv DATA_FIN
                    -------      |   |   -------
    +---------+     snd DATA_FIN /   \ snd DATA_ACK[DFIN] +---------+
    |  M_FIN  |<-----------------       ------------------>| M_CLOSE |
    | WAIT-1  |--------------------------                  |  WAIT   |
    +---------+          rcv DATA_FIN \                    +---------+
      | rcv DATA_ACK[DFIN]    -------  |                     M_CLOSE  |
      | --------------   snd DATA_ACK  |                     -------  |
      | CLOSE all subflows             |                 snd DATA_FIN |
      V                                V                             V
    +----------+             +----------+                  +----------+
    |M_FINWAIT-2|            | M_CLOSING |                  | M_LAST-ACK|
    +----------+             +----------+                  +----------+
      |             rcv DATA_ACK[DFIN] |         rcv DATA_ACK[DFIN] |
      | rcv DATA_FIN    -------------- |         -------------- |
      |  -------      CLOSE all subflows |       CLOSE all subflows |
      | snd DATA_ACK[DFIN]             V        delete MPTCP PCB  V
       \                      +----------+                +---------+
        ----------------------->|M_TIME WAIT|---------------->| M_CLOSED|
                              +----------+                +---------+
                                    All subflows in CLOSED
                                        ------------
                                      delete MPTCP PCB
```

                Figure 17: Finite State Machine for Connection Closure

Authors' Addresses

   Alan Ford
   Cisco
   Ruscombe Business Park
   Ruscombe, Berkshire RG10 9NN
   UK

   EMail: alanford@cisco.com

   Costin Raiciu
   University Politehnica of Bucharest
   Splaiul Independentei 313
   Bucharest
   Romania

   EMail: costin.raiciu@cs.pub.ro

   Mark Handley
   University College London
   Gower Street
   London WC1E 6BT
   UK

   EMail: m.handley@cs.ucl.ac.uk

   Olivier Bonaventure
   Universite catholique de Louvain
   Pl. Ste Barbe, 2
   Louvain-la-Neuve 1348
   Belgium

   EMail: olivier.bonaventure@uclouvain.be