Internet Engineering Task Force   A. Ford
Internet-Draft   Roke Manor Research
Intended status: Informational   C. Raiciu
Expires: July 25, 2011   M. Handley
                                             University College London
                                                              S. Barre
                                                   Universite catholique de
                                                                Louvain
                                                            J. Iyengar
                                            Franklin and Marshall College
                                                       January 21, 2011

          Architectural Guidelines for Multipath TCP Development
                     draft-ietf-mptcp-architecture-05

Abstract

   Hosts are often connected by multiple paths, but TCP restricts
   communications to a single path per transport connection. Resource
   usage within the network would be more efficient were these multiple
   paths able to be used concurrently. This should enhance user
   experience through improved resilience to network failure and higher
   throughput.

   This document outlines architectural guidelines for the development
   of a Multipath Transport Protocol, with references to how these
   architectural components come together in the development of a
   Multipath TCP protocol. This document lists certain high level
   design decisions that provide foundations for the design of the MPTCP
   protocol, based upon these architectural requirements.

Copyright Notice

Table of Contents

1. Introduction

   As the Internet evolves, demands on Internet resources are ever-
   increasing, but often these resources (in particular, bandwidth)
   cannot be fully utilised due to protocol constraints both on the end-
   systems and within the network. If these resources could be used
   concurrently, end user experience could be greatly improved. Such
   enhancements would also reduce the necessary expenditure on network
   infrastructure that would otherwise be needed to create an equivalent
   improvement in user experience. By the application of resource
   pooling [3], these available resources can be 'pooled' such that they
   appear as a single logical resource to the user.

   Multipath transport aims to realize some of the goals of resource
   pooling by simultaneously making use of multiple disjoint (or
   partially disjoint) paths across a network. The two key benefits of
   multipath transport are:

   o To increase the resilience of the connectivity by providing
      multiple paths, protecting end hosts from the failure of one.

   o To increase the efficiency of the resource usage, and thus
      increase the network capacity available to end hosts.

   Multipath TCP is a modified version of TCP [1] that implements a
   multipath transport and achieves these goals by pooling multiple
   paths within a transport connection, transparently to the
   application. Multipath TCP is primarily concerned with utilising
   multiple paths end-to-end, where one or both end host is multi-homed.
   It may also have applications where multiple paths exist within the
   network and can be manipulated by an end host, such as using
   different port numbers with ECMP [4].

   MPTCP, defined in [5], is a specific protocol that instantiates the
   Multipath TCP concept. This document looks both at general
   architectural principles for a Multipath TCP fulfilling the goals
   described in Section 2, as well as the key design decisions behind
   MPTCP, which are detailed in Section 5.

   Although multihoming and multipath functions are not new to transport
   protocols (SCTP [6] being a notable example), MPTCP aims to gain
   wide-scale deployment by recognising the importance of application
   and network compatibility goals. These goals, discussed in detail in
   Section 2, relate to the appearance of MPTCP to the network (so non-
   MPTCP-aware entities see it as TCP) and to the application (through
   providing an service equivalent to TCP for non-MPTCP-aware
   applications).

This document has three key purposes: (i) it describes goals for a
multipath transport - goals that MPTCP is designed to meet; (ii) it
lays out an architectural basis for MPTCP's design - a discussion
that applies to other multipath transports as well; and (iii) it
discusses and documents high-level design decisions made in MPTCP's
development, and considers their implications.

Companion documents to this architectural overview are those which
provide details of the protocol extensions [5], congestion control
algorithms [7], and application-level considerations [8]. Put
together, these components specify a complete Multipath TCP design.
We note that specific components are replaceable in accordance with
the layer and functional decompositions discussed in this document.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
"SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
document are to be interpreted as described in RFC 2119 [2].

1.2. Terminology

Regular/Single-Path TCP: The standard version of the TCP [1]
   protocol in use today, operating between a single pair of IP
   addresses.

Multipath TCP: A modified version of the TCP protocol that supports
   the simultaneous use of multiple paths between hosts.

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

Host: An end host either initiating or terminating a Multipath TCP
   connection.

MPTCP: The proposed protocol extensions specified in [5] to provide
   a Multipath TCP implementation.

Subflow: A flow of TCP segments operating over an individual path,
   which forms part of a larger Multipath TCP connection.

(Multipath TCP) Connection: A set of one or more subflows combined
   to provide a single Multipath TCP service to an application at a
   host.

1.3. Reference Scenario

   The diagram shown in Figure 1 illustrates a typical usage scenario
   for Multipath TCP. Two hosts, A and B, are communicating with each
   other. These hosts are multi-homed and multi-addressed, providing
   two disjoint connections to the Internet. The addresses on each host
   are referred to as A1, A2, B1 and B2. There are therefore up to four
   different paths between the two hosts: A1-B1, A1-B2, A2-B1, A2-B2.

```
      +------+              _____              +------+
      |      |A1 _____   (          ) _____  B1|      |
      | Host |--/         (          )         \--| Host |
      |      |           (   Internet   )          |      |
      |  A   |--_____ (          )_____/--|  B   |
      |      |A2          (_____)         B2|      |
      +------+                                   +------+
```
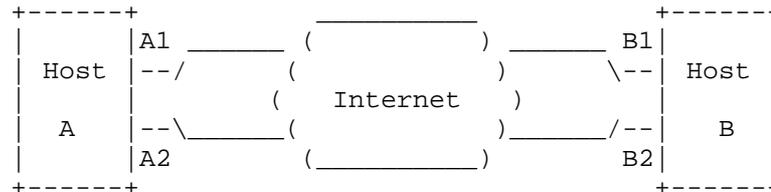
                Figure 1: Simple Multipath TCP Usage Scenario

   The scenario could have any number of addresses (1 or more) on each
   host, as long as the number of paths available between the two hosts
   is 2 or more (i.e. num_addr(A) * num_addr(B) > 1). The paths created
   by these address combinations through the Internet need not be
   entirely disjoint - potential fairness issues introduced by shared
   bottlenecks need to be handled by the Multipath TCP congestion
   controller. Furthermore, the paths through the Internet often do not
   provide a pure end-to-end service, and instead may be affected by
   middleboxes such as NATs and Firewalls.

2. Goals

   This section outlines primary goals that Multipath TCP aims to meet.
   These are broadly broken down into: functional goals, which steer
   services and features that Multipath TCP must provide; and
   compatibility goals, which determine how Multipath TCP should appear
   to entities that interact with it.

2.1. Functional Goals

   In supporting the use of multiple paths, Multipath TCP has the
   following two functional goals.

   o Improve Throughput: Multipath TCP MUST support the concurrent use
      of multiple paths. To meet the minimum performance incentives for
      deployment, a Multipath TCP connection over multiple paths SHOULD
      achieve no lesser throughput than a single TCP connection over the
      best constituent path.

   o Improve Resilience: Multipath TCP MUST support the use of multiple
      paths interchangeably for resilience purposes, by permitting
      segments to be sent and re-sent on any available path. It follows
      that, in the worst case, the protocol MUST be no less resilient
      than regular single-path TCP.

   As distribution of traffic among available paths and responses to
   congestion are done in accordance with resource pooling principles
   [3], a secondary effect of meeting these goals is that widespread use
   of Multipath TCP over the Internet should improve overall network
   utility by shifting load away from congested bottlenecks and by
   taking advantage of spare capacity wherever possible.

   Furthermore, Multipath TCP SHOULD feature automatic negotiation of
   its use. A host supporting Multipath TCP that requires the other
   host to do so too must be able to detect reliably whether this host
   does in fact support the required extensions, using them if so, and
   otherwise automatically falling back to single-path TCP.

2.2. Compatibility Goals

   In addition to the functional goals listed above, a Multipath TCP
   must meet a number of compatibility goals in order to support
   deployment in today's Internet. These goals fall into the following
   categories:

2.2.1. Application Compatibility

   Application compatibility refers to the appearance of Multipath TCP
   to the application both in terms of the API that can be used and the
   expected service model that is provided.

   Multipath TCP MUST follow the same service model as TCP [1]: in-
   order, reliable, and byte-oriented delivery. Furthermore, a
   Multipath TCP connection SHOULD provide the application with no worse
   throughput or resilience than it would expect from running a single
   TCP connection over any one of its available paths. A Multipath TCP
   may not, however, be able to provide the same level of consistency of
   throughput and latency as a single TCP connection. These, and other,
   application considerations are discussed in detail in [8].

   A multipath-capable equivalent of TCP MUST retain some level of
   backward compatibility with existing TCP APIs, so that existing
   applications can use the newer transport merely by upgrading the
   operating systems of the end-hosts. This does not preclude the use
   of an advanced API to permit multipath-aware applications to specify
   preferences, nor for users to configure their systems in a different
   way from the default, for example switching on or off the automatic

use of multipath extensions.

It is possible for regular TCP sessions today to survive brief breaks
in connectivity by retaining state at end hosts before a timeout
occurs. It would be desirable to support similar session continuity
in MPTCP, however the circumstances could be different. Whilst in
regular TCP the IP addresses will remain constant across the break in
connectivity, in MPTCP a different interface may appear. It is
desirable (but not mandated) to support this kind of "break-before-
make" session continuity. This places constraints on security
mechanisms, however, as discussed in Section 5.8. Timeouts for this
function would be locally configured.

2.2.2. Network Compatibility

In the traditional Internet architecture, network devices operate at
the network layer and lower layers, with the layers above the network
layer instantiated only at the end-hosts. While this architecture,
shown in Figure 2, was initially largely adhered to, this layering no
longer reflects the "ground truth" in the Internet with the
proliferation of middleboxes [9]. Middleboxes routinely interpose on
the transport layer; sometimes even completely terminating transport
connections, thus leaving the application layer as the first real
end-to-end layer, as shown in Figure 3.

```
+-------------+                                      +-------------+
| Application |<------------ end-to-end ------------>| Application |
+-------------+                                      +-------------+
|  Transport  |<------------ end-to-end ------------>|  Transport  |
+-------------+  +-------------+  +-------------+     +-------------+
|   Network   |<->|   Network   |<->|   Network   |<->|   Network   |
+-------------+  +-------------+  +-------------+     +-------------+
    End Host          Router           Router           End Host
```
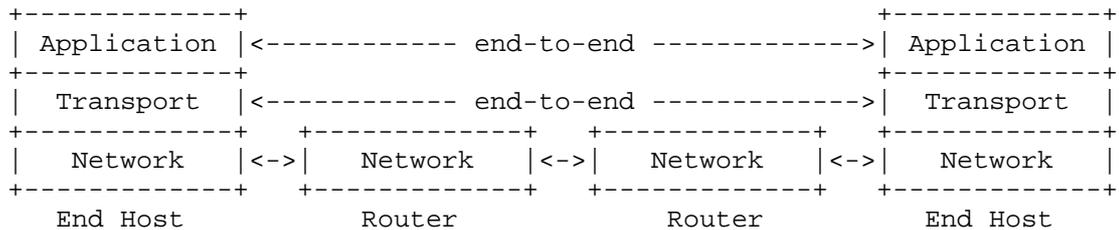
Figure 2: Traditional Internet Architecture

```
+-------------+                                      +-------------+
| Application |<------------ end-to-end ------------>| Application |
+-------------+                    +-------------+   +-------------+
|  Transport  |<------------------>|  Transport  |<->|  Transport  |
+-------------+  +-------------+    +-------------+   +-------------+
|   Network   |<->|   Network   |<->|   Network   |<->|   Network   |
+-------------+  +-------------+    +-------------+   +-------------+
                                      Firewall,
    End Host          Router         NAT, or Proxy     End Host
```
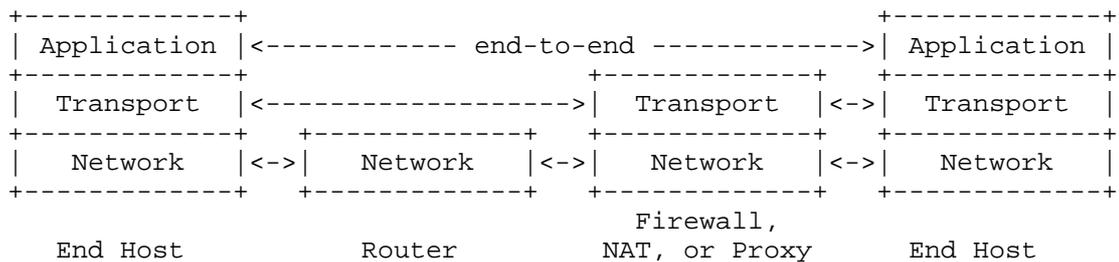
Figure 3: Internet Reality

Middleboxes that interpose on the transport layer result in loss of "fate-sharing" [10], that is, they often hold "hard" state that, when lost or corrupted, results in loss or corruption of the end-to-end transport connection.

The network compatibility goal requires that the multipath extension to TCP retains compatibility with the Internet as it exists today, including making reasonable efforts to be able to traverse predominant middleboxes such as firewalls, NATs, and performance enhancing proxies [9]. This requirement comes from recognizing middleboxes as a significant deployment bottleneck for any transport that is not TCP or UDP, and constrains Multipath TCP to appear as TCP does on the wire and to use established TCP extensions where necessary. To ensure end-to-endness of the transport, we further require Multipath TCP to preserve fate-sharing without making any assumptions about middlebox behavior.

A detailed analysis of middlebox behaviour and the impact on the Multipath TCP architecture is presented in Section 7. In addition, network compatibility must be retained to the extent that Multipath TCP MUST fall back to regular TCP if there are insurmountable incompatibilities for the multipath extension on a path.

Middleboxes may also cause some TCP features to be able to exist on one subflow but not another. Typically these will be at the subflow level (such as SACK [11]) and thus do not affect the connection-level behaviour. In the future, any proposed TCP connection-level extensions should consider how they can co-exist with MPTCP.

The modifications to support Multipath TCP remain at the transport layer, although some knowledge of the underlying network layer is required. Multipath TCP SHOULD work with IPv4 and IPv6 interchangeably, i.e. one connection may operate over both IPv4 and IPv6 networks.

2.2.3. Compatibility with other network users

As a corollary to both network and application compatibility, the architecture must enable new Multipath TCP flows to coexist gracefully with existing single-path TCP flows, competing for bandwidth neither unduly aggressively nor unduly timidly (unless low-precedence operation is specifically requested by the application, such as with LEDBAT). The use of multiple paths MUST NOT unduly harm users using single-path TCP at shared bottlenecks, beyond the impact that would occur from another single-path TCP flow. Multiple Multipath TCP flows on a shared bottleneck MUST share bandwidth between each other with similar fairness to that which occurs at a shared bottleneck with single-path TCP.

2.3. Security Goals

   The extension of TCP with multipath capabilities will bring with it a
   number of new threats, analysed in detail in [12]. The security goal
   for Multipath TCP is to provide a service no less secure than
   regular, single-path TCP. This will be achieved through a
   combination of existing TCP security mechanisms (potentially modified
   to align with the Multipath TCP extensions) and of protection against
   the new multipath threats identified. The design decisions derived
   from this goal are presented in Section 5.8.

2.4. Related Protocols

   There are several similarities between SCTP [6] and MPTCP, in that
   both can make use of multiple addresses at end hosts to give some
   multi-path capability. In SCTP, the primary use case is to support
   redundancy and mobility for multihomed hosts (i.e. a single path will
   change one of its end host addresses); the simultaneous use of
   multiple paths is not supported . Extensions are proposed to support
   simultaneous multipath transport [13], but these are yet to be
   standardised. By far the most widely used stream-based transport
   protocol is, however, TCP [1], and SCTP does not meet the network and
   application compatibility goals specified in Section 2.2. For
   network compatibility, there are issues with various middleboxes
   (especially NATs) that are unaware of SCTP and consequently end up
   blocking it. For application compatibility, applications need to
   actively choose to use SCTP, and with the deployment issues very few
   choose to do so. MPTCP's compatibility goals are in part based on
   these observations of SCTP's deployment issues.

3. An Architectural Basis For Multipath TCP

   We now present one possible transport architecture that we believe
   can effectively support the goals for Multipath TCP. The new
   Internet model described here is based on ideas proposed earlier in
   Tng ("Transport next-generation") [14]. While by no means the only
   possible architecture supporting multipath transport, Tng
   incorporates many lessons learned from previous transport research
   and development practice, and offers a strong starting point from
   which to consider the extant Internet architecture and its bearing on
   the design of any new Internet transports or transport extensions.

```
           +-----------------+
           |   Application   |
           +-----------------+  ^ Application-oriented transport
           |                 |  | functions (Semantic Layer)
           + - - Transport - -+ ---------------------------------
           |                 |  | Network-oriented transport
           +-----------------+  v functions (Flow+Endpoint Layer)
           |     Network     |
           +-----------------+
             Existing Layers           Tng Decomposition
```
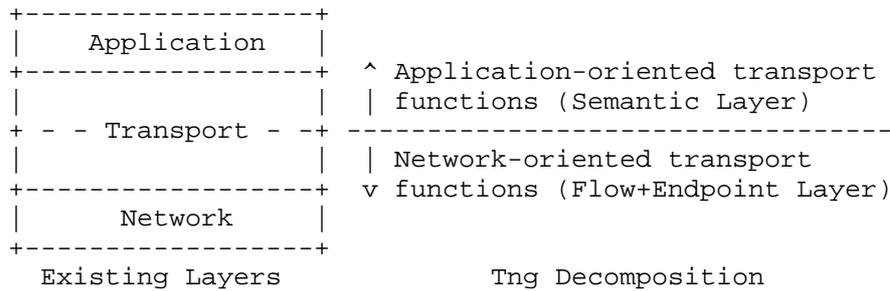
                 Figure 4: Decomposition of Transport Functions

   Tng loosely splits the transport layer into "application-oriented"
   and "network-oriented" layers, as shown in Figure 4. The
   application-oriented "Semantic" layer implements functions driven
   primarily by concerns of supporting and protecting the application's
   end-to-end communication, while the network-oriented "Flow+Endpoint"
   layer implements functions such as endpoint identification (using
   port numbers) and congestion control. These network-oriented
   functions, while traditionally located in the ostensibly "end-to-end"
   Transport layer, have proven in practice to be of great concern to
   network operators and the middleboxes they deploy in the network to
   enforce network usage policies [15] [16] or optimize communication
   performance [17]. Figure 5 shows how middleboxes interact with
   different layers in this decomposed model of the transport layer: the
   application-oriented layer operates end-to-end, while the network-
   oriented layer operates "segment-by-segment" and can be interposed
   upon by middleboxes.

```
+-------------+                                      +-------------+
| Application |<------------ end-to-end ------------>| Application |
+-------------+                                      +-------------+
|  Semantic   |<------------ end-to-end ------------>|  Semantic   |
+-------------+   +-------------+   +-------------+   +-------------+
|Flow+Endpoint|<->|Flow+Endpoint|<->|Flow+Endpoint|<->|Flow+Endpoint|
+-------------+   +-------------+   +-------------+   +-------------+
|   Network   |<->|   Network   |<->|   Network   |<->|   Network   |
+-------------+   +-------------+   +-------------+   +-------------+
                     Firewall         Performance
     End Host        or NAT        Enhancing Proxy       End Host
```
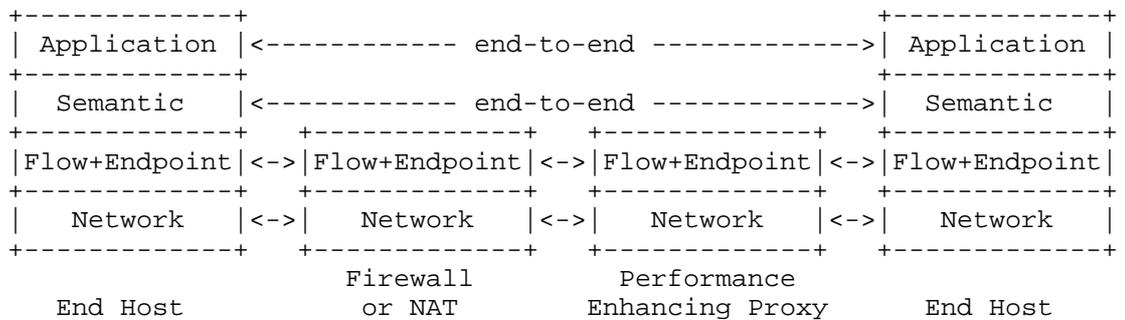
                 Figure 5: Middleboxes in the new Internet model

   MPTCP's architectural design follows Tng's decomposition as shown in
   Figure 6. MPTCP, which provides application compatibility through
   the preservation of TCP-like semantics of global ordering of
   application data and reliability, is an instantiation of the

"application-oriented" Semantic layer; whereas the subflow TCP
component, which provides network compatibility by appearing and
behaving as a TCP flow in the network, is an instantiation of the
"network-oriented" Flow+Endpoint layer.

```
+------------------------+   +------------------------------+
|      Application       |   |         Application          |
+------------------------+   +------------------------------+
|        Semantic        |   |            MPTCP             |
|-----------+------------| + - - - - - - + - - - - - - - +
| Flow+Endpt | Flow+Endpt |   | Subflow (TCP) | Subflow (TCP) |
+-----------+------------+   +--------------+---------------+
|  Network  |   Network  |   |      IP      |      IP       |
+-----------+------------+   +--------------+---------------+
```
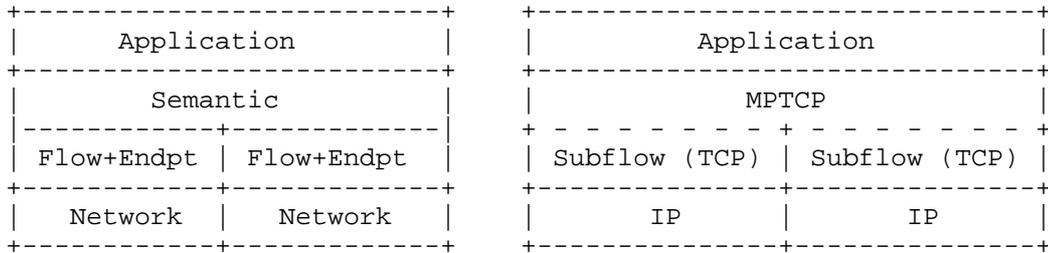
Figure 6: Relationship between Tng (left) and MPTCP (right)

As a protocol extension to TCP, MPTCP thus explicitly acknowledges
middleboxes in its design, and specifies a protocol that operates at
two scales: the MPTCP component operates end-to-end, while it allows
the TCP component to operate segment-by-segment.

4. A Functional Decomposition of MPTCP

The previous two sections have discussed the goals for a Multipath
TCP design, and provided a basis for decomposing the functions of a
transport protocol in order to better understand the form a solution
should take. This section builds upon this analysis by presenting
the functional components that are used within the MPTCP design.

MPTCP makes use of (what appear to the network to be) standard TCP
sessions, termed "subflows", to provide the underlying transport per
path, and as such these retain the network compatibility desired.
MPTCP-specific information is carried in a TCP-compatible manner,
although this mechanism is separate from the actual information being
transferred so could evolve in future revisions. Figure 7
illustrates the layered architecture.

```
                                   +------------------------------+
                                   |          Application         |
          +---------------+        +------------------------------+
          |  Application  |        |             MPTCP            |
          +---------------+        + - - - - - - + - - - - - - - +
          |      TCP      |        | Subflow (TCP) | Subflow (TCP) |
          +---------------+        +------------------------------+
          |      IP       |        |      IP       |      IP      |
          +---------------+        +------------------------------+
```
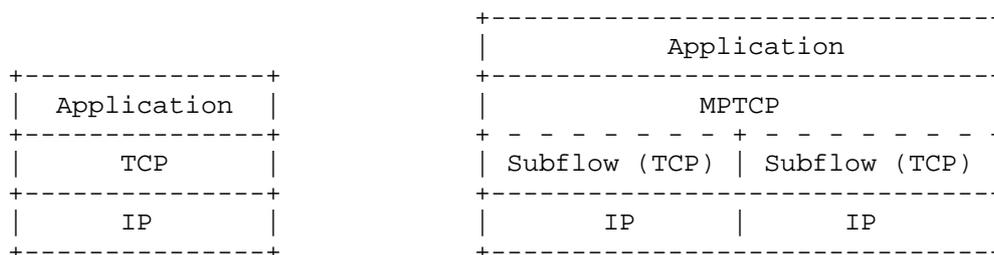
       Figure 7: Comparison of Standard TCP and MPTCP Protocol Stacks

   Situated below the application, the MPTCP extension in turn manages
   multiple TCP subflows below it. In order to do this, it must
   implement the following functions:

   o Path Management: This is the function to detect and use multiple
      paths between two hosts. MPTCP uses the presence of multiple IP
      addresses at one or both of the hosts as an indicator of this.
      The path management features of the MPTCP protocol are the
      mechanisms to signal alternative addresses to hosts, and
      mechanisms to set up new subflows joined to an existing MPTCP
      connection.

   o Packet Scheduling: This function breaks the bytestream received
      from the application into segments to be transmitted on one of the
      available subflows. The MPTCP design makes use of a data sequence
      mapping, associating segments sent on different subflows to a
      connection-level sequence numbering, thus allowing segments sent
      on different subflows to be correctly re-ordered at the receiver.
      The packet scheduler is dependent upon information about the
      availability of paths exposed by the path management component,
      and then makes use of the subflows to transmit queued segments.
      This function is also responsible for connection-level re-ordering
      on receipt of packets from the TCP subflows, according to the
      attached data sequence mappings.

   o Subflow (single-path TCP) Interface: A subflow component takes
      segments from the packet-scheduling component and transmits them
      over the specified path, ensuring detectable delivery to the host.
      MPTCP uses TCP underneath for network compatibility; TCP ensures
      in-order, reliable delivery. TCP adds its own sequence numbers to
      the segments; these are used to detect and retransmit lost packets
      at the subflow layer. On receipt, the subflow passes its
      reassembled data to the packet scheduling component for
      connection-level reassembly; the data sequence mapping from the
      sender's packet scheduling component allows re-ordering of the
      entire bytestream.

   o Congestion Control: This function coordinates congestion control
      across the subflows. As specified, this congestion control
      algorithm MUST ensure that a MPTCP connection does not unfairly
      take more bandwidth than a single path TCP flow would take at a
      shared bottleneck. An algorithm to support this is specified in
      [7].

   These functions fit together as follows. The Path Management looks
   after the discovery (and if necessary, initialisation) of multiple
   paths between two hosts. The Packet Scheduler then receives a stream
   of data from the application destined for the network, and undertakes
   the necessary operations on it (such as segmenting the data into
   connection-level segments, and adding a connection-level sequence
   number) before sending it on to a subflow. The subflow then adds its
   own sequence number, ACKs, and passes them to network. The receiving
   subflow re-orders data (if necessary) and passes it to the packet
   scheduling component, which performs connection level re-ordering,
   and sends the data stream to the application. Finally, the
   congestion control component exists as part of the packet scheduling,
   in order to schedule which segments should be sent at what rate on
   which subflow.

5. High-Level Design Decisions

   There is seemingly a wide range of choices when designing a multipath
   extension to TCP. However, the goals as discussed earlier in this
   document constrain the possible solutions, leaving relative little
   choice in many areas. Here, we outline high-level design choices
   that draw from the architectural basis discussed earlier in
   Section 3, which the design of MPTCP [5] takes into account.

5.1. Sequence Numbering

   MPTCP uses two levels of sequence spaces: a connection level sequence
   number, and another sequence number for each subflow. This permits
   connection-level segmentation and reassembly, and retransmission of
   the same part of connection-level sequence space on different
   subflow-level sequence space.

   The alternative approach would be to use a single connection level
   sequence number, which gets sent on multiple subflows. This has two
   problems: first, the individual subflows will appear to the network
   as TCP sessions with gaps in the sequence space; this in turn may
   upset certain middleboxes such as intrusion detection systems, or
   certain transparent proxies, and would thus go against the network
   compatibility goal. Second, the sender would not be able to
   attribute packet losses or receptions to the correct path when the

same segment is sent on multiple paths (i.e. in the case of
retransmissions).

The sender must be able to tell the receiver how to reassemble the
data, for delivery to the application. In order to achieve this, the
receiver must determine how subflow-level data (carrying subflow
sequence numbers) maps at the connection level. We refer to this as
the Data Sequence Mapping. This mapping takes the form (data seq,
subflow seq, length), i.e. for a given number of bytes (the length),
the subflow sequence space beginning at the given sequence number
maps to the connection-level sequence space (beginning at the given
data seq number). This information could conceivably have various
sources.

One option to signal the Data Sequence Mapping would be to use
existing fields in the TCP segment (such as subflow seqno, length)
and only add the data sequence number to each segment, for instance
as a TCP option. This would be vulnerable, however, to middleboxes
that resegment or assemble data, since there is no specified
behaviour for coalescing TCP options. If one signalled (data seqno,
length), this would still be vulnerable to middleboxes that coalesce
segments and do not understand MPTCP signalling so do not correctly
rewrite the options.

Because of these potential issues, the design decision taken in the
MPTCP protocol is that whenever a mapping for subflow data needs to
be conveyed to the other host, all three pieces of data (data seq,
subflow seq, length) must be sent. To reduce the overhead, it would
be permissible for the mapping to be sent periodically and cover more
than a single segment. Further experimentation is required to
determine what tradeoffs exist regarding the frequency at which
mappings should be sent. It could also be excluded entirely in the
case of a connection before more than one subflow is used, where the
data-level and subflow-level sequence space is the same.

5.2. Reliability and Retransmissions

MPTCP features acknowledgements at connection-level as well as
subflow-level acknowledgements, in order to provide a robust service
to the application.

Under normal behaviour, MPTCP can use the data sequence mapping and
subflow ACKs to decide when a connection-level segment was received.
The transmission of TCP ACKs for a subflow are handled entirely at
the subflow level, in order to maintain TCP semantics and trigger
subflow-level retransmissions. This has certain implications on end-
to-end semantics. It means that once a segment is ACKed at the
subflow level it cannot be discarded in the re-order buffer at the

connection level. Secondly, unlike in standard TCP, a receiver
cannot simply drop out-of-order segments if needed (for instance, due
to memory pressure). Under certain circumstances, therefore, it may
be desirable to drop segments after acknowledgement on the subflow
but before delivery to the application, and this can be facilitated
by a connection-level acknowledgement.

Furthermore, it is possible to conceive of some cases where
connection-level acknowledgements could improve robustness. Consider
a subflow traversing a transparent proxy: if the proxy ACKs a segment
and then crashes, the sender will not retransmit the lost segment on
another subflow, as it thinks the segment has been received. The
connection grinds to a halt despite having other working subflows,
and the sender would be unable to determine the cause of the problem.
An example situation where this may occur would be mobility between
wireless access points, each of which operates a transport-level
proxy. Finally, as an optimisation, it may be feasible for a
connection-level acknowledgement to be transmitted over the shortest
Round-Trip Time (RTT) path, potentially reducing send buffer
requirements (see Section 5.3).

Therefore, to provide a fully robust multipath TCP solution given the
above constraints, MPTCP for use on the public Internet MUST feature
explicit connection-level acknowledgements, in addition to subflow-
level acknowledgements. A connection-level acknowledgement would
only be required in order to signal when the receive window moves
forward; the heuristics for using such a signal are discussed in more
detail in the protocol specification [5].

Regarding retransmissions, it MUST be possible for a segments to be
retransmitted on a different subflow to that on which it was
originally sent. This is one of MPTCP's core goals, in order to
maintain integrity during temporary or permanent subflow failure, and
this is enabled by the dual sequence number space.

The scheduling of retransmissions will have significant impact on
MPTCP user experience. The current MPTCP specification suggests that
data outstanding on subflows that have timed out should be
rescheduled for transmission on different subflows. This behaviour
aims to minimize disruption when a path breaks, and uses the first
timeout as indicators. More conservative versions would be to use
second or third timeouts for the same segment.

Typically, fast retransmit on an individual subflow will not trigger
retransmission on another subflow, although this may still be
desirable in certain cases, for instance to reduce the receive buffer
requirements. However, in all cases with retransmissions on
different subflows, the lost segments SHOULD still be sent on the

path that lost them. This is currently believed to be necessary to
maintain subflow integrity, as per the network compatibility goal.
By doing this, some efficiency is lost, and it is unclear at this
point what the optimal retransmit strategy is.

Large-scale experiments are therefore required in order to determine
the most appropriate retransmission strategy, and recommendations
will be refined once more information is available.

5.3. Buffers

To ensure in-order delivery, MPTCP must use a connection level
receive buffer, where segments are placed until they are in order and
can be read by the application.

In regular, single-path TCP, it is usually recommended to set the
receive buffer to 2*BDP (Bandwidth-Delay Product, i.e. BDP = BW*RTT,
where BW = Bandwidth and RTT = Round-Trip Time). One BDP allows
supporting reordering of segments by the network. The other BDP
allows the connection to continue during fast retransmit: when a
segment is fast retransmitted, the receiver must be able to store
incoming data during one more RTT.

For MPTCP, the story is a bit more complicated. The ultimate goal is
that a subflow packet loss or subflow failure should not affect the
throughput of other working subflows; the receiver should have enough
buffering to store all data until the missing segment is re-
transmitted and reaches the destination.

The worst case scenario would be when the subflow with the highest
RTT/RTO (Round-Trip Time or Retransmission TimeOut) experiences a
timeout; in that case the receiver has to buffer data from all
subflows for the duration of the RTO. Thus, the smallest connection-
level receive buffer that would be needed to avoid stalling with
subflow failures is sum(BW_i)*RTO_max, where BW_i = Bandwidth for
each subflow and RTO_max is the largest RTO across all subflows.

This is an order of magnitude more than the receive buffer required
for a single connection, and is probably too expensive for practical
purposes. A more sensible requirement is to avoid stalls in the
absence of timeouts. Therefore, the RECOMMENDED receive buffer is
2*sum(BW_i)*RTT_max, where RTT_max is the largest RTT across all
subflows. This buffer sizing ensures subflows do not stall when fast
retransmit is triggered on any subflow.

The resulting buffer size should be small enough for practical use.
However, there may be extreme cases where fast, high throughput paths
(e.g. 100Mb/s, 10ms RTT) are used in conjunction with slow paths

(e.g. 1Mb/s, 1000ms RTT). In that case the required receive buffer
would be 12.5MB, which is likely too big. In extreme cases such as
this example, it may be prudent to only use some of the fastest
available paths for the MPTCP connection, potentially using the slow
path(s) for backup only.

Send Buffer: The RECOMMENDED send buffer is the same size as the
recommended receive buffer i.e., 2*sum(BW_i)*RTT_max. This is
because the sender must store locally the segments sent but
unacknowledged by the connection level ACK. The send buffer size
matters particularly for hosts that maintain a large number of
ongoing connections. If the required send buffer is too large, a
host can choose to only send data on the fast subflows, using the
slow subflows only in cases of failure.

5.4. Signalling

Since MPTCP uses TCP as its subflow transport mechanism, a MPTCP
connection will also begin as a single TCP connection. Nevertheless,
it must signal to the peer that it supports MPTCP and wishes to use
it on this connection. As such, a TCP Option will be used to
transmit this information, since this is the established mechanism
for indicating additional functionality on a TCP session.

In addition, further signalling is required during the operation of a
MPTCP session, such as that for reassembly for multiple subflows, and
for informing the other host about potential other available
addresses.

The MPTCP protocol design will, however, use TCP Options for this
additional signalling. This has been chosen as the mechanism most
fitting in with the goals as specified in Section 2. With this
mechanism, the signalling requires to operate MPTCP is transported
separately from the data, allowing it to be created and processed
separately from the data stream, and retaining architectural
compatibility with network entities.

This decision is the consensus of the Working Group (following
detailed discussions at IETF78), and the main reasons for this are as
follows:

o TCP options are the traditional signalling method for TCP;

o A TCP option on a SYN is the most compatible way for an end host
   to signal it is MPTCP-capable;

o If connection-level ACKs are signalled in the payload then they
   may suffer from packet loss and may be congestion-controlled,

which may affect the data throughput in the forward direction and
could lead to head-of-line blocking;

o Middleboxes, such as NAT traversal helpers, can easily parse TCP
   options, e. g., to rewrite addresses.

On the other hand, the main drawbacks of TCP options compared to TLV
encoding in the payload are:

o There is limited space for signalling messages;

o A middlebox may, potentially, drop a packet with an unknown
   option;

o The transport of control information in options is not necessarily
   reliable.

The detailed design of MPTCP alleviates these issues as far as
possible by carefully considering the size of MPTCP options, and
seamlessly falling back to regular TCP on the loss of control data.

Both option and payload encoding may interfere with offloading of TCP
processing to high speed network interface cards, such as
segmentation, checksumming, and reassembly. For network cards
supporting MPTCP, signalling in TCP options should simplify
offloading due to the separate handling of MPTCP signalling and data.

5.5. Path Management

Currently, the network does not expose path diversity between pairs
of IP addresses. In order to achieve path diversity from today's IP
networks, in the typical case MPTCP uses multiple addresses at one or
both hosts to infer different paths across the network. It is
expected that these paths, whilst not necessarily entirely non-
overlapping, will be sufficiently disjoint to allow multipath to
achieve improved throughput and robustness. The use of multiple IP
addresses is a simple mechanism that requires no additional features
in the network.

Multiple different (source, destination) address pairs will thus be
used as path selectors in most cases. Each path will be identified
by a standard five-tuple (i.e. source address, destination address,
source port, destination port, protocol), however, which can allow
the extension of MPTCP to use ports as well as addresses as path
selectors. This will allow hosts to use port-based load balancing
with MPTCP, for example if the network routes different ports over
different paths (which may be the case with technologies such as
Equal Cost MultiPath (ECMP) routing [4]). It should be noted,

however, that ISPs often undertake traffic engineering in order to
optimise resource utilisation within their networks, and care should
be taken (by both ISPs and developers) that MPTCP using broadly
similar paths does not adversely interfere with this.

For increased chance of successfully setting up additional subflows
(such as when one end is behind a firewall, NAT, or other restrictive
middlebox), either host SHOULD be able to add new subflows to a MPTCP
connection. MPTCP MUST be able to handle paths that appear and
disappear during the lifetime of a connection (for example, through
the activation of an additional network interface).

The path management is a separate function from the packet
scheduling, subflow interface, and congestion control functions of
MPTCP, as documented in Section 4. As such it would be feasible to
replace this IP-address-based design with an alternative path
selection mechanism in the future, with no significant changes to the
other functional components.

5.6. Connection Identification

Since a MPTCP connection may not be bound to a traditional 5-tuple
(source address and port, destination address and port, protocol
number) for the entirety of its existence, it is desirable to provide
a new mechanism for connection identification. This will be useful
for MPTCP-aware applications, and for the MPTCP implementation (and
MPTCP-aware middleboxes) to have a unique identifier with which to
associate the multiple subflows.

Therefore, each MPTCP connection requires a connection identifier at
each host, which is locally unique within that host. In many ways,
this is analogous to an ephemeral port number in regular TCP. The
manifestation and purpose of such an identifier is out of the scope
of this architecture document.

Legacy applications will not, however, have access to this identifier
and in such cases a MPTCP connection will be identified by the
5-tuple of the first TCP subflow. It is out of the scope of this
document, however, to define the behaviour of the MPTCP
implementation if the first TCP subflow later fails. If there are
MPTCP-unaware applications that make assumptions about continued
existence of the initial address pair, their behaviour could be
disrupted by carrying on regardless. It is expected that this is a
very small, possibly negligible, set of applications, however. MPTCP
MUST NOT be used for applications that request to bind to a specific
address or interface, since such applications are making a deliberate
choice of path in use.

   Since the requirements of applications are not clear at this stage,
   however, it is as yet unconfirmed whether carrying on in the event of
   the loss of the initial address pair would be a damaging assumption
   to make. This behaviour will be an implementation-specific solution,
   and as such it is expected to be chosen by implementors once more
   research has been undertaken to determine its impact.

5.7. Congestion Control

   As discussed in network-layer compatibility requirements
   Section 2.2.3, there are three goals for the congestion control
   algorithms used by a MPTCP implementation: improve throughput (at
   least as well as a single-path TCP connection would perform); do no
   harm to other network users (do not take up more capacity on any one
   path than if it was a single path flow using only that route - this
   is particularly relevant for shared bottlenecks); and balance
   congestion by moving traffic away from the most congested paths. To
   achieve these goals, the congestion control algorithms on each
   subflow must be coupled in some way. A proposal for a suitable
   congestion control algorithm is given in [7].

5.8. Security

   A detailed threat analysis for Multipath TCP is presented in a
   separate document [12]. This focuses on flooding attacks and
   hijacking attacks that can be launched against a Multipath TCP
   connection.

   The basic security goal of Multipath TCP, as introduced in
   Section 2.3, can be stated as: "provide a solution that is no worse
   than standard TCP".

   From the threat analysis, and with this goal in mind, three key
   security requirements can be identified. A multi-addressed Multipath
   TCP SHOULD be able to:

   o Provide a mechanism to confirm that the parties in a subflow
      handshake are the same as in the original connection setup (e.g.
      require use of a key exchanged in the initial handshake in the
      subflow handshake, to limit the scope for hijacking attacks).

   o Provide verification that the peer can receive traffic at a new
      address before adding it (i.e. verify that the address belongs to
      the other host, to prevent flooding attacks).

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

Additional mechanisms have been deployed as part of standard TCP
stacks to provide resistance to Denial-of-Service attacks. For
example, there are various mechanisms to protect against TCP reset
attacks [18], and Multipath TCP should continue to support similar
protection. In addition, TCP SYN Cookies [19] were developed to
allow a TCP server to defer the creation of session state in the
SYN_RCVD state, and remain stateless until the ESTABLISHED state had
been reached. Multipath TCP should, ideally, continue to provide
such functionality and, at a minimum, avoid significant computational
burden prior to reaching the ESTABLISHED state (of the Multipath TCP
connection as a whole).

It should be noted that aspects of the Multipath TCP design space
place constraints on the security solution:

o The use of TCP options significantly limits the amount of
   information that can be carried in the handshake.

o The need to work through middleboxes results in the need to handle
   mutability of packets.

o The desire to support a 'break-before-make' (as well as a 'make-
   before-break') approach to adding subflows (within a limited time
   period) implies that a host cannot rely on using a pre-existing
   subflow to support the addition of a new one.

The MPTCP protocol will be designed with these security requirements
in mind, and the protocol specification [5] will document how these
are met.

6. Software Interactions

6.1. Interactions with Applications

In the case of applications that have used an existing API call to
bind to a specific address or interface, the MPTCP extension MUST NOT
be used. This is because the applications are indicating a clear
choice of path to use and thus will have expectations of behaviour
that must be maintained, in order to adhere to the application
compatibility goals.

Interactions with applications are presented in [8] - including, but
not limited to, performances changes that may be expected, semantic
changes, and new features that may be requested through an enhanced
API.

TCP features the ability to send "Urgent" data, the delivery of which

to the application may or may not be out-of-band. The use of this
feature is not recommended due to security implications and
implementation differences [20]. MPTCP requires contiguous data to
support its Data Sequence Mapping over multiple segments, and
therefore the Urgent pointer cannot interrupt an existing mapping.
An MPTCP implementation MAY choose to support sending Urgent data,
and if it does, it SHOULD send the Urgent data on the soonest
available unassigned subflow sequence space. Incoming Urgent data
SHOULD be mapped to connection-level sequence space and delivered to
the application analogous to Urgent data in regular TCP.

6.2. Interactions with Management Systems

To enable interactions between TCP and network management systems,
the TCP [21] and TCP Extended Statistics (ESTATS) [22] MIBs have been
defined. MPTCP should share the these MIBs for aspects that are
designed to be transparent to the application.

It is anticipated that a MPTCP MIB will be defined in the future,
once experience of experimental MPTCP deployments is gathered. This
MIB would provide access to MPTCP-specific properties such as whether
MPTCP is enabled, and the number and properties of the individual
paths in use.

7. Interactions with Middleboxes

As discussed in Section 2.2, it is a goal of MPTCP to be deployable
today and thus compatible with the majority of middleboxes. This
section summarises the issues that may arise with NATs, firewalls,
proxies, intrusion detection systems, and other middleboxes that, if
not considered in the protocol design, may hinder its deployment.

This section is intended primarily as a description of options and
considerations only. Protocol-specific solutions to these issues
will be given in the companion documents.

Multipath TCP will be deployed in a network that no longer provides
just basic datagram delivery. A myriad of middleboxes are deployed
to optimize various perceived problems with the Internet protocols:
NATs primarily address IP address space shortage [15], Performance
Enhancing Proxies (PEPs) optimize TCP for different link
characteristics [17], firewalls [16] and intrusion detection systems
try to block malicious content from reaching a host, and traffic
normalizers [23] ensure a consistent view of the traffic stream to
Intrusion Detection Systems (IDS) and hosts.

All these middleboxes optimize current applications at the expense of

future applications. In effect, future applications will often need
to behave in a similar fashion to existing ones, in order to increase
the chances of successful deployment. Further, the precise behaviour
of all these middleboxes is not clearly specified, and implementation
errors make matters worse, raising the bar for the deployment of new
technologies.

The following list of middlebox classes documents behaviour that
could impact the use of MPTCP. This list is used in [5] to describe
the features of the MPTCP protocol that are used to mitigate the
impact of these middlebox behaviours.

o NATs: Network Address Translators decouple the host's local IP
   address (and, in the case of NAPTs, port) with that which is seen
   in the wider Internet when the packets are transmitted through a
   NAT. This adds complexity, and reduces the chances of success,
   when signalling IP addresses.

o PEPs: Performance Enhancing Proxies, which aim to improve the
   performance of protocols over low-performance (e.g. high latency
   or high error rate) links. As such, they may "split" a TCP
   connection and behaviour such as proactive ACKing may occur, and
   therefore it is no longer guaranteed that one host is
   communicating directly with another. PEPs, firewalls or other
   middleboxes may also change the declared receive window size.

o Traffic Normalizers: These aim to eliminate ambiguities and
   potential attacks at the network level, and amongst other things
   are unlikely to permit holes in TCP-level sequence space (which
   has impact on MPTCP's retransmission and subflow sequence
   numbering design choices).

o Firewalls: on top of preventing incoming connections, firewalls
   may also attempt additional protection such as sequence number
   randomization (so a sender cannot reliably know what TCP sequence
   number the receiver will see).

o Intrusion Detection Systems: IDSs may look for traffic patterns to
   protect a network, and may have false positives with MPTCP and
   drop the connections during normal operation. Future MPTCP-aware
   middleboxes will require the ability to correlate the various
   paths in use.

o Content-aware Firewalls: Some middleboxes may actively change data
   in packets, such as re-writing URIs in HTTP traffic.

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

o TCP Options: some middleboxes may drop packets with unknown TCP
   options, or strip those options from the packets.

o Segmentation and Coalescing: middleboxes (or even something as
   close to the end host as TCP Segmentation Offloading (TSO) on a
   Network Interface Card (NIC)) may change the packet boundaries
   from those which the sender intended. It may do this by splitting
   packets, or coalescing them together. This leads to two major
   impacts: we cannot guarantee where a packet boundary will be, and
   we cannot say for sure what a middlebox will do with TCP options
   in these cases (they may be repeated, dropped, or sent only once).

8. Contributors

   The authors would like to acknowledge the contributions of Andrew
   McDonald and Bryan Ford to this document.

   The authors would also like to thank the following people for
   detailed reviews: Olivier Bonaventure, Gorry Fairhurst, Iljitsch van
   Beijnum, Philip Eardley, Michael Scharf, Lars Eggert, Cullen
   Jennings, Joel Halpern, Juergen Quittek, Alexey Melnikov, David
   Harrington, Jari Arkko and Stewart Bryant.

9. Acknowledgements

   Alan Ford, Costin Raiciu, Mark Handley, and Sebastien Barre are
   supported by Trilogy (http://www.trilogy-project.org), a research
   project (ICT-216372) partially funded by the European Community under
   its Seventh Framework Program. The views expressed here are those of
   the author(s) only. The European Commission is not liable for any
   use that may be made of the information in this document.

10. IANA Considerations

   None.

11. Security Considerations

   This informational document provides an architectural overview for
   Multipath TCP and so does not, in itself, raise any security issues.
   A separate threat analysis [12] lists threats that can exist with a
   Multipath TCP. However, a protocol based on the architecture in this
   document will have a number of security requirements. The high level
   goals for such a protocol are identified in Section 2.3, whilst

Section 5.8 provides more detailed discussion of security
requirements and design decisions which are applied in the MPTCP
protocol design [5].

12. References

12.1. Normative References

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

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

12.2. Informative References

[3]   Wischik, D., Handley, M., and M. Bagnulo Braun, "The Resource
      Pooling Principle", ACM SIGCOMM CCR vol. 38 num. 5, pp. 47-52,
      October 2008,
      <http://ccr.sigcomm.org/online/files/p47-handleyA4.pdf>.

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

[5]   Ford, A., Raiciu, C., and M. Handley, "TCP Extensions for
      Multipath Operation with Multiple Addresses",
      draft-ietf-mptcp-multiaddressed-02 (work in progress),
      October 2010.

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

[7]   Raiciu, C., Handley, M., and D. Wischik, "Coupled Congestion
      Control for Multipath Transport Protocols",
      draft-ietf-mptcp-congestion-01 (work in progress),
      January 2011.

[8]   Scharf, M. and A. Ford, "MPTCP Application Interface
      Considerations", draft-ietf-mptcp-api-00 (work in progress),
      November 2010.

[9]   Carpenter, B. and S. Brim, "Middleboxes: Taxonomy and Issues",
      RFC 3234, February 2002.

[10] Carpenter, B., "Internet Transparency", RFC 2775,
      February 2000.

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

   [12]   Bagnulo, M., "Threat Analysis for TCP Extensions for Multi-path
          Operation with Multiple Addresses", draft-ietf-mptcp-threat-07
          (work in progress), January 2011.

   [13]   Becke, M., Dreibholz, T., Iyengar, J., Natarajan, P., and M.
          Tuexen, "Load Sharing for the Stream Control Transmission
          Protocol (SCTP)", draft-tuexen-tsvwg-sctp-multipath-01 (work in
          progress), December 2010.

   [14]   Ford, B. and J. Iyengar, "Breaking Up the Transport Logjam",
           ACM HotNets, October 2008.

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

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

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

   [18]   Ramaiah, A., Stewart, R., and M. Dalal, "Improving TCP's
          Robustness to Blind In-Window Attacks", RFC 5961, August 2010.

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

   [20]   Gont, F. and A. Yourtchenko, "On the Implementation of the TCP
          Urgent Mechanism", RFC 6093, January 2011.

   [21]   Raghunarayan, R., "Management Information Base for the
          Transmission Control Protocol (TCP)", RFC 4022, March 2005.

   [22]   Mathis, M., Heffner, J., and R. Raghunarayan, "TCP Extended
          Statistics MIB", RFC 4898, May 2007.

   [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>.

Appendix A. Changelog

   (For removal by the RFC Editor)

A.1. Changes since draft-ietf-mptcp-architecture-04

   o Responded to IETF Last Call and IESG review comments.

A.2. Changes since draft-ietf-mptcp-architecture-03

   o Responded to AD review comments.

A.3. Changes since draft-ietf-mptcp-architecture-02

   o Responded to WG last call review comments. Included editorial
      fixes, adding Section 2.4, and improving Section 5.4 and
      Section 7.

A.4. Changes since draft-ietf-mptcp-architecture-01

   o Responded to review comments.

   o Added security sections.

A.5. Changes since draft-ietf-mptcp-architecture-00

   o Added middlebox compatibility discussion (Section 7).

   o Clarified path identification (TCP 4-tuple) in Section 5.5.

   o Added brief scenario and diagram to Section 1.3.

Authors' Addresses

   Alan Ford
   Roke Manor Research
   Old Salisbury Lane
   Romsey, Hampshire SO51 0ZN
   UK

   Phone: +44 1794 833 465
   Email: alan.ford@roke.co.uk

Costin Raiciu
University College London
Gower Street
London WC1E 6BT
UK

Email: c.raiciu@cs.ucl.ac.uk

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

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

Sebastien Barre
Universite catholique de Louvain
Pl. Ste Barbe, 2
Louvain-la-Neuve 1348
Belgium

Phone: +32 10 47 91 03
Email: sebastien.barre@uclouvain.be

Janardhan Iyengar
Franklin and Marshall College
Mathematics and Computer Science
PO Box 3003
Lancaster, PA 17604-3003
USA

Phone: 717-358-4774
Email: jiyengar@fandm.edu

Coupled Congestion Control for Multipath Transport Protocols
draft-ietf-mptcp-congestion-07

Abstract

   Often endpoints are connected by multiple paths, but communications
   are usually restricted to a single path per connection. Resource
   usage within the network would be more efficient were it possible for
   these multiple paths to be used concurrently. Multipath TCP is a
   proposal to achieve multipath transport in TCP.

   New congestion control algorithms are needed for multipath transport
   protocols such as Multipath TCP, as single path algorithms have a
   series of issues in the multipath context. One of the prominent
   problems is that running existing algorithms such as standard TCP
   independently on each path would give the multipath flow more than
   its fair share at a bottleneck link traversed by more than one of its
   subflows. Further, it is desirable that a source with multiple paths
   available will transfer more traffic using the least congested of the
   paths, achieving a property called resource pooling where a bundle of
   links effectively behaves like one shared link with bigger-capacity.
   This would increase the overall efficiency of the network and also
   its robustness to failure.

   This document presents a congestion control algorithm which couples
   the congestion control algorithms running on different subflows by
   linking their increase functions, and dynamically controls the
   overall aggressiveness of the multipath flow. The result is a
   practical algorithm that is fair to TCP at bottlenecks while moving
   traffic away from congested links.

Status of this Memo

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

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF). Note that other groups may also distribute
   working documents as Internet-Drafts. The list of current Internet-

Drafts is at http://datatracker.ietf.org/drafts/current/.

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

This Internet-Draft will expire on January 30, 2012.

Copyright Notice

Table of Contents

1. Requirements Language

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

2. Introduction

   Multipath TCP (MPTCP, [I-D.ietf-mptcp-multiaddressed]) is a set of
   extensions to regular TCP [RFC0793] that allows one TCP connection to
   be spread across multiple paths. MPTCP distributes load through the
   creation of separate "subflows" across potentially disjoint paths.

   How should congestion control be performed for multipath TCP? First,
   each subflow must have its own congestion control state (i.e. cwnd)
   so that capacity on that path is matched by offered load. The
   simplest way to achieve this goal is to simply run standard TCP
   congestion control on each subflow. However this solution is
   unsatisfactory as it gives the multipath flow an unfair share when
   the paths taken by its different subflows share a common bottleneck.

   Bottleneck fairness is just one requirement multipath congestion
   control should meet. The following three goals capture the desirable
   properties of a practical multipath congestion control algorithm:

   o Goal 1 (Improve Throughput) A multipath flow should perform at
      least as well as a single path flow would on the best of the paths
      available to it.

   o Goal 2 (Do no harm) A multipath flow should not take up more
      capacity from any of the resources shared by its different paths,
      than if it was a single flow using only one of these paths. This
      guarantees it will not unduly harm other flows.

   o Goal 3 (Balance congestion) A multipath flow should move as much
      traffic as possible off its most congested paths, subject to
      meeting the first two goals.

   Goals 1 and 2 together ensure fairness at the bottleneck. Goal 3
   captures the concept of resource pooling [WISCHIK]: if each multipath
   flow sends more data through its least congested path, the traffic in
   the network will move away from congested areas. This improves
   robustness and overall throughput, among other things. The way to
   achieve resource pooling is to effectively "couple" the congestion
   control loops for the different subflows.

   We propose an algorithm that couples the additive increase function

of the subflows, and uses unmodified TCP behavior in case of a drop.
The algorithm relies on the traditional TCP mechanisms to detect
drops, to retransmit data, etc.

Detecting shared bottlenecks reliably is quite difficult, but is just
one part of a bigger question. This bigger question is how much
bandwidth a multipath user should use in total, even if there is no
shared bottleneck.

The congestion controller aims to set the multipath flow's aggregate
bandwidth to be the same as a regular TCP flow would get on the best
path available to the multipath flow. To estimate the bandwidth of a
regular TCP flow, the multipath flow estimates loss rates and round
trip times and computes the target rate. Then it adjusts the overall
aggresiveness (parameter alpha) to achieve the desired rate.

While the mechanism above applies always, its effect depends on
whether the multipath TCP flow influences or does not influence the
link loss rates (low vs. high statistical multiplexing). If MPTCP
does not influence link loss rates, MPTCP will get the same
throughput as TCP on the best path. In cases with low statistical
multiplexing, where the multipath flow influences the loss rates on
the path, the multipath throughput will be strictly higher than a
single TCP would get on any of the paths. In particular, if using
two idle paths, multipath throughput will be sum of the two paths'
throughput.

This algorithm ensures bottleneck fairness and fairness in the
broader, network sense. We acknowledge that current TCP fairness
criteria are far from ideal, but a multipath TCP needs to be
deployable in the current Internet. If needed, new fairness criteria
can be implemented by the same algorithm we propose by appropriately
scaling the overall aggressiveness.

It is intended that the algorithm presented here can be applied to
other multipath transport protocols, such as alternative multipath
extensions to TCP, or indeed any other congestion-aware transport
protocols. However, for the purposes of example this document will,
where appropriate, refer to the MPTCP protocol.

The design decisions and evaluation of the congestion control
algorithm are published in [NSDI].

The algorithm presented here only extends standard TCP congestion
control for multipath operation. It is foreseeable that other
congestion controllers will be implemented for multipath transport to
achieve the bandwidth-scaling properties of the newer congestion
control algorithms for regular TCP (such as Compound TCP and Cubic).

3. Coupled Congestion Control Algorithm

   The algorithm we present only applies to the increase phase of the
   congestion avoidance state specifying how the window inflates upon
   receiving an ack. The slow start, fast retransmit, and fast recovery
   algorithms, as well as the multiplicative decrease of the congestion
   avoidance state are the same as in standard TCP[RFC5681].

   Let cwnd_i be the congestion window on the subflow i. Let tot_cwnd
   be the sum of the congestion windows of all subflows in the
   connection. Let p_i, rtt_i and mss_i be the loss rate, round trip
   time (i.e. smoothed round trip time estimate used by TCP) and maximum
   segment size on subflow i.

   We assume throughout this document that the congestion window is
   maintained in bytes, unless otherwise specified. We briefly describe
   the algorithm for packet-based implementations of cwnd in section
   Section 4.2.

   Our proposed "Linked Increases" algorithm MUST:

   o For each ack received on subflow i, increase cwnd_i by

$$\min\left(\frac{\text{alpha} * \text{bytes\_acked} * \text{mss\_i}}{\text{tot\_cwnd}} , \frac{\text{bytes\_acked} * \text{mss\_i}}{\text{cwnd\_i}} \right) \quad (1)$$

   The increase formula (1) takes the minimum between the computed
   increase for the multipath subflow (first argument to min), and the
   increase TCP would get in the same scenario (the second argument).
   In this way, we ensure that any multipath subflow cannot be more
   aggressive than a TCP flow in the same circumstances, hence achieving
   goal 2 (do no harm).

   "alpha" is a parameter of the algorithm that describes the
   aggresiveness of the multipath flow. To meet Goal 1 (improve
   throughput), the value of alpha is chosen such that the aggregate
   throughput of the multipath flow is equal to the throughput a TCP
   flow would get if it ran on the best path.

   To get an intuition of what the algorithm is trying to do, let's take
   the case where all the subflows have the same round trip time and
   MSS. In this case the algorithm will grow the total window by
   approximately alpha*MSS per RTT. This increase is distributed to the
   individual flows according to their instantaneous window size.
   Subflow i will increase by alpha*cwnd_i/tot_cwnd segments per RTT.

   Note that, as in standard TCP, when tot_cwnd is large the increase

may be 0. In this case the increase MUST be set to 1. We discuss
how to implement this formula in practice in the next section.

We assume implementations use an approach similar to appropriate byte
counting (ABC, [RFC3465]), where the bytes_acked variable records the
number of bytes newly acknowledged. If this is not the case,
bytes_acked SHOULD be set to mss_i.

To compute tot_cwnd, it is an easy mistake to sum up cwnd_i across
all subflows: when a flow is in fast retransmit, its cwnd is
typically inflated and no longer represents the real congestion
window. The correct behavior is to use the ssthresh value for flows
in fast retransmit when computing tot_cwnd. To cater for connections
that are app limited, the computation should consider the minimum
between flight_size_i and cwnd_i, and flight_size_i and ssthresh_i
where appropriate.

The total throughput of a multipath flow depends on the value of
alpha and the loss rates, maximum segment sizes and round trip times
of its paths. Since we require that the total throughput is no worse
than the throughput a single TCP would get on the best path, it is
impossible to choose a-priori a single value of alpha that achieves
the desired throughput in every occasion. Hence, alpha must be
computed based on the observed properties of the paths.

The formula to compute alpha is:

$$
alpha = tot\_cwnd * \frac{\max_i \dfrac{cwnd\_i}{rtt\_i^2}}{\left(sum_i \dfrac{cwnd\_i}{rtt\_i} \right)^2} \qquad (2)
$$

The formula (2) is derived by equalizing the rate of the multipath
flow with the rate of a TCP running on the best path, and solving for
alpha.

4. Implementation Considerations

Equation (2) implies that alpha is a floating point value. This
would require performing costly floating point operations whenever an
ACK is received, Further, in many kernels floating point operations
are disabled. There is an easy way to approximate the above

calculations using integer arithmetic.

4.1. Computing alpha in Practice

Let alpha_scale be an integer. When computing alpha, use alpha_scale
* tot_cwnd instead of tot_cwnd, and do all the operations in integer
arithmetic.

Then, scale down the increase per ack by alpha_scale. The resulting
algorithm is a simple change from Equation (1):

o For each ack received on subflow i, increase cwnd_i by

$$\min \left(\frac{alpha * bytes\_acked * mss\_i}{alpha\_scale * tot\_cwnd} , \frac{bytes\_acked * mss\_i}{cwnd\_i} \right) \quad (3)$$

The alpha_scale parameter denotes the precision we want for computing
alpha. Observe that the errors in computing the numerator or the
denominator in the formula for alpha are quite small, as the cwnd in
bytes is typically much larger than the RTT (measured in ms).

With these changes, all the operations can be done using integer
arithmetic. We propose alpha_scale be a small power of two, to allow
using faster shift operations instead of multiplication and division.
Our experiments show that using alpha_scale=512 works well in a wide
range of scenarios. Increasing alpha_scale increases precision, but
also increases the risk of overflow when computing alpha. Using
64bit operations would solve this issue. Another option is to
dynamically adjust alpha_scale when computing alpha; in this way we
avoid overflow and obtain maximum precision.

It is possible to implement the algorithm by calculating tot_cwnd on
each ack, however this would be costly especially when the number of
subflows is large. To avoid this overhead the implementation MAY
choose to maintain a new per connection state variable called
tot_cwnd. If it does so, the implementation will update tot_cwnd
value whenever the individual subflows' windows are updated.
Updating only requires one more addition or subtraction operation
compared to the regular, per subflow congestion control code, so its
performance impact should be minimal.

Computing alpha per ack is also costly. We propose alpha to be a per
connection variable, computed whenever there is a drop and once per
RTT otherwise. More specifically, let cwnd_new be the new value of
the congestion window after it is inflated or after a drop. Update
alpha only if the quotient of cwnd_i/mss_i differs from the quotient
of cwnd_new_i/mss_i.

In certain cases with small RTTs, computing alpha can still be
expensive. We observe that if RTTs were constant, it is sufficient
to compute alpha once per drop, as alpha does not change between
drops (the insight here is that cwnd_i/cwnd_j = constant as long as
both windows increase). Experimental results show that even if round
trip times are not constant, using average round trip time per
sawtooth instead of instantaneous round trip time (i.e. TCP's
smoothed RTT estimator) gives good precision for computing alpha.
Hence, it is possible to compute alpha only once per drop using a
modified version of equation (2) where rtt_i is replaced with
rtt_avg_i.

If using average round trip time, rtt_avg_i will be computed by
sampling the rtt_i whenever the window can accommodate one more
packet, i.e. when cwnd / mss < (cwnd+increase)/mss. The samples are
averaged once per sawtooth into rtt_avg_i. This sampling ensures
that there is no sampling bias for larger windows.

Given tot_cwnd and alpha, the congestion control algorithm is run for
each subflow independently, with similar complexity to the standard
TCP increase code [RFC5681].

4.2. Implementation Considerations when CWND is Expressed in Packets

   When the congestion control algorithm maintains cwnd in packets
   rather than bytes, the algorithms above must change to take into
   account path mss.

   To compute the increase when an ack is received, the implementation
   for multipath congestion control is a simple extension of the
   standard TCP code. In standard TCP cwnd_cnt is an additional state
   variable that tracks the number of segments acked since the last cwnd
   increment; cwnd is incremented only when cwnd_cnt > cwnd; then
   cwnd_cnt is set to 0.

   In the multipath case, cwnd_cnt_i is maintained for each subflow as
   above, and cwnd_i is increased by 1 when cwnd_cnt_i > max(alpha_scale
   * tot_cwnd / alpha, cwnd_i).

   When computing alpha for packet-based stacks, the errors in computing
   the terms in the denominator are larger (this is because cwnd is much
   smaller and rtt may be comparatively large). Let max be the index of
   the subflow used in the numerator. To reduce errors, it is easiest
   to move rtt_max (once calculated) from the numerator to the
   denominator, changing equation (2) to obtain the equivalent formula
   below.

$$alpha = alpha\_scale * tot\_cwnd * \frac{cwnd\_max}{\left(\sum_i \frac{rtt\_max * cwnd\_i}{rtt\_i} \right)^2} \quad (4)$$

Note that the calculation of alpha does not take into account path
mss, and is the same for stacks that keep cwnd in bytes or packets.
With this formula, the algorithm for computing alpha will match the
rate of TCP on the best path in B/s for byte-oriented stacks, and in
packets/s in packet-based stacks. In practice, mss rarely changes
between paths so this shouldn't be a problem.

However, it is simple to derive formulae allowing packet-based stacks
to achieve byte rate fairness (and viceversa) if needed. In
particular, for packet-based stacks wanting byte-rate fairness,
equation (4) above changes as follows: cwnd_max is replaced by
cwnd_max * mss_max * mss_max, while cwnd_i is replaced with cwnd_i *
mss_i.

5. Discussion

The algorithm we've presented fully achieves Goals 1 and 2, but does
not achieve full resource pooling (Goal 3). Resource pooling
requires that no traffic should be transferred on links with higher
loss rates. To achieve perfect resource pooling, one must couple
both increase and decrease of congestion windows across subflows, as
in [KELLY].

There are a few problems with such a fully-coupled controller.
First, it will probe insufficiently paths with high loss rates, and
will fail to detect free capacity when it becomes available. Second,
such controllers tend to exhibit "flappiness": when the paths have
similar levels of congestion, the congestion controller will tend to
allocate all the window to one random subflow, and allocate zero
window to the other subflows. The controller will perform random
flips between these stable points. This doesn't seem desirable in
general, and is particularly bad when the achieved rates depend on
the RTT (as in the current Internet): in such a case, the resulting
rate with fluctuate unpredictably depending on which state the
controller is in, hence violating Goal 1.

By only coupling increases our proposal probes high-loss paths,
detecting free capacity quicker. Our proposal does not suffer from
flappiness but also achieves less resource pooling. The algorithm
will allocate window to the subflows such that p_i * cwnd_i =

constant, for all i. Thus, when the loss rates of the subflows are
equal, each subflow will get an equal window, removing flappiness.
When the loss rates differ, progressively more window will be
allocated to the flow with the lower loss rate. In contrast, perfect
resource pooling requires that all the window should be allocated on
the path with the lowest loss rate. Further details can be found in
[NSDI].

6. Security Considerations

One security concern relates to what we call the traffic-shifting
attack: on-path attackers can drop packets belonging to a multipath
subflow, which in turn makes the path seem congested and will force
the sender's congestion controller to avoid that path and push more
data over alternate subflows.

The attacker's goal is to create congestion on the corresponding
alternative paths. This behaviour is entirely feasible, but will
only have minor effects: by design, the coupled congestion controller
is less (or similarly) aggressive on any of its paths than a single
TCP flow. Thus, the biggest effect this attack can have is to make a
multipath subflow be as aggressive as a single TCP flow.

Another effect of the traffic-shifting attack is that the new path
can monitor all the traffic, whereas before it could only see a
subset of traffic. We believe that if privacy is needed, splitting
traffic across multiple paths with MPTCP is not the right solution in
the first place; end-to-end encryption should be used instead.

Besides the traffic-shifting attack mentioned above, the coupled
congestion control algorithm defined in this draft adds no other
security considerations to those found in
[I-D.ietf-mptcp-multiaddressed] and [RFC6181]. Detailed security
analysis for the Multipath TCP protocol itself is included in
[I-D.ietf-mptcp-multiaddressed] and [RFC6181].

7. Acknowledgements

8. IANA Considerations

   This document does not require any action from IANA.

9. References

9.1. Normative References

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

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

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

9.2. Informative References

   [I-D.ietf-mptcp-multiaddressed]
              Ford, A., Raiciu, C., Handley, M., and O. Bonaventure,
              "TCP Extensions for Multipath Operation with Multiple
              Addresses", draft-ietf-mptcp-multiaddressed-04 (work in
              progress), July 2011.

   [KELLY]   Kelly, F. and T. Voice, "Stability of end-to-end
              algorithms for joint routing and rate control", ACM
              SIGCOMM CCR vol. 35 num. 2, pp. 5-12, 2005,
              <http://portal.acm.org/citation.cfm?id=1064415>.

   [NSDI]   Wischik, D., Raiciu, C., Greenhalgh, A., and M. Handley,
              "Design, Implementation and Evaluation of Congestion
              Control for Multipath TCP", Usenix NSDI , March 2011, <htt
              p://www.cs.ucl.ac.uk/staff/c.raiciu/files/mptcp-nsdi.pdf>.

   [RFC3465] Allman, M., "TCP Congestion Control with Appropriate Byte
              Counting (ABC)", RFC 3465, February 2003.

   [RFC6181] Bagnulo, M., "Threat Analysis for TCP Extensions for
              Multipath Operation with Multiple Addresses", RFC 6181,
              March 2011.

   [WISCHIK] Wischik, D., Handley, M., and M. Bagnulo Braun, "The
              Resource Pooling Principle", ACM SIGCOMM CCR vol. 38 num.
              5, pp. 47-52, October 2008,
              <http://ccr.sigcomm.org/online/files/p47-handleyA4.pdf>.

Authors' Addresses

    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

    Damon Wischik
    University College London
    Gower Street
    London WC1E 6BT
    UK

    Email: d.wischik@cs.ucl.ac.uk

Internet Engineering Task Force   A. Ford
Internet-Draft   Cisco
Intended status: Experimental   C. Raiciu
Expires: April 25, 2013   University Politehnica of
                                                           Bucharest
                                                          M. Handley
                                            University College London
                                                        O. Bonaventure
                                               Universite catholique de
                                                             Louvain
                                                    October 22, 2012

TCP Extensions for Multipath Operation with Multiple Addresses
draft-ietf-mptcp-multiaddressed-12

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 provides the components necessary to establish and
   use multiple TCP flows across potentially disjoint paths.

Copyright Notice

Table of Contents

1. Introduction

   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], presenting a safe congestion control
      algorithm for coupling the behaviour of the multiple paths in
      order to "do no harm" to other network users.

   o Application Considerations [6], discussing what impact MPTCP will
      have on applications, what applications will want to do with
      MPTCP, and as a consequence of these factors, what API extensions
      an MPTCP implementation should present.

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
   utilisation 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 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.

Fall-back: The protocol should be able to fall back to standard TCP
   with no interference from the user, to be able to communicate with
   legacy hosts.

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 behaviour 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].

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       |
          +---------------+    +------------------------------+
```
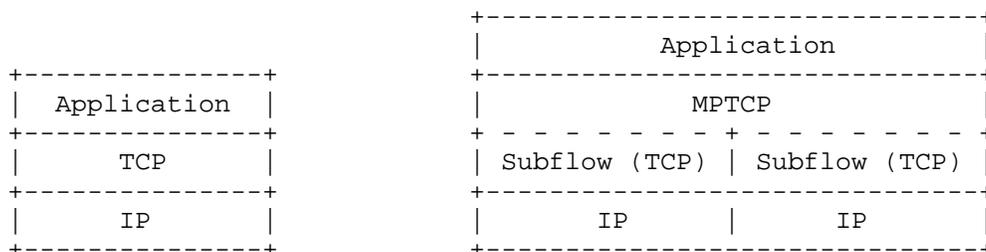
        Figure 1: Comparison of Standard TCP and MPTCP Protocol Stacks

1.3. Terminology

   This document makes use of a number of terms which 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 similarly 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: A 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 is
      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)            |
              |               |---------------->|               |
              |               |<----------------|               |
              |               |                 |               |
              |               |                 |               |
```
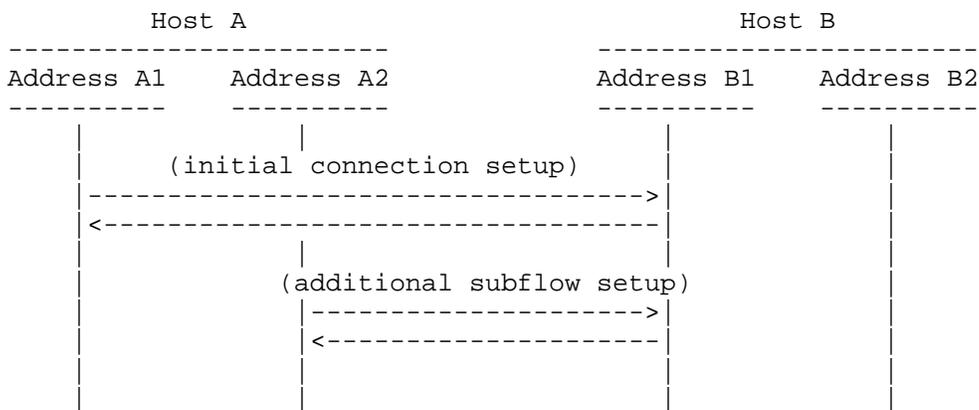
                    Figure 2: Example MPTCP Usage Scenario

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 which follows in Section 3.

   A Multipath TCP connection provides a bidirectionnal 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 subflows 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;
and 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 setup. 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 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 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 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 signalled 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 bits 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 acknowledgement (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 acknowledgement 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 being
   only 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 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 behaviour 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 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 [8], the following steps are
      taken: keys are sent in the clear in the MP_CAPABLE messages;
      MP_JOIN messages are secured with HMAC-SHA1 ([9], [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 signalled using optional TCP header fields.
   A single TCP option number ("Kind") will be assigned by IANA for
   MPTCP (see Section 8), and then individual messages will be
   determined by a "sub-type", the values of which will also be 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 sub-type value of the
symbolic name as defined in Section 8. This sub-type is a four-bit
field - the first four 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)                     |
   +-------------------------------------------------------------+
```
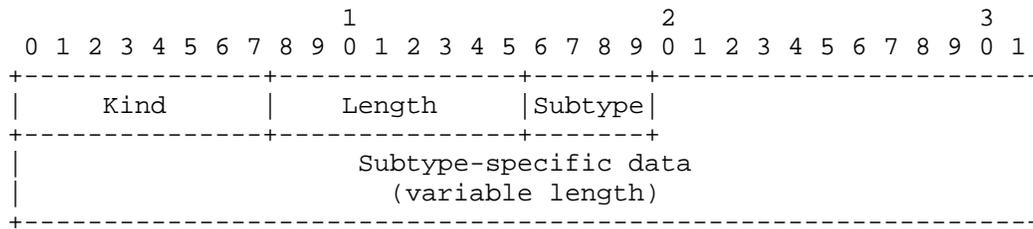
Figure 3: MPTCP option format

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 [10]) 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 [11] in regular TCP. Therefore, an MPTCP
implementation receiving a duplicate ACK which 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 Acknowledgement Number are within window) MUST be
undertaken before processing any MPTCP signals, as described in [12].

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 which 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 [13]. 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 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)                  |
   |                                                               |
   +---------------------------------------------------------------+
```
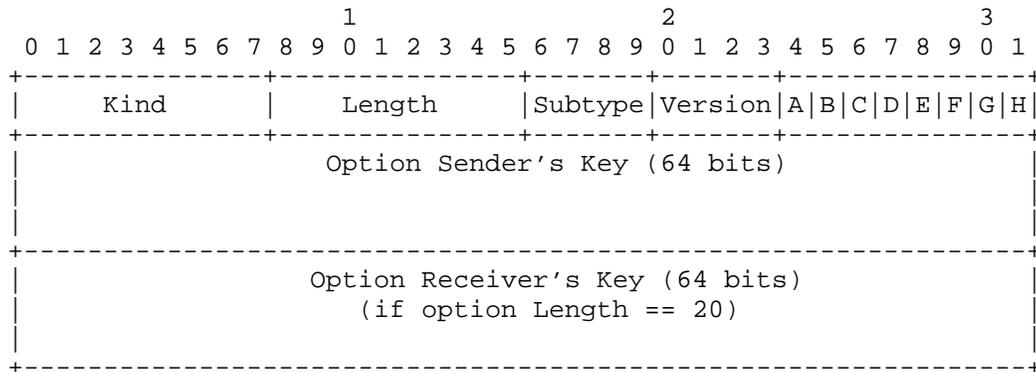
              Figure 4: Multipath Capable (MP_CAPABLE) option

The first four 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 four bits of this octet
specifies 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, labelled "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, labelled "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, labelled "C" through "H", are used
   for crypto algorithm negotiation. Currently only the rightmost
   bit, labelled "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. 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], [14]) 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 (labelled "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
one bit set - this is the chosen algorithm. The rationale for this
behaviour 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 fall-back 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 and 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 re-ordered 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 (IDSN) on a 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 behaviour 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],
   [14]) 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 not only sends 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 [13].

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 a 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, labelled '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)               |
       +---------------------------------------------------------------+
```
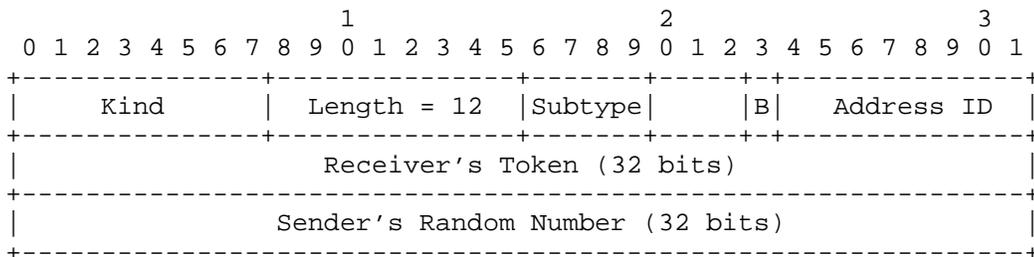
    Figure 5: Join Connection (MP_JOIN) option (for initial SYN)

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 an RST,
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 and 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 [9] is used, along with the SHA-1 hash
algorithm [4] (potentially implemented as in [14]), 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, and 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 and 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)               |
       +---------------------------------------------------------------+
```
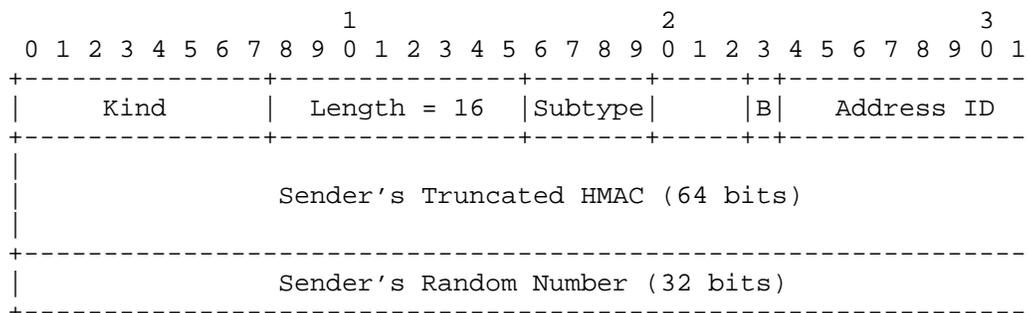
Figure 6: Join Connection (MP_JOIN) option (for responding SYN/ACK)

```
                            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)                    |
       |                                                               |
       |                                                               |
       +---------------------------------------------------------------+
```
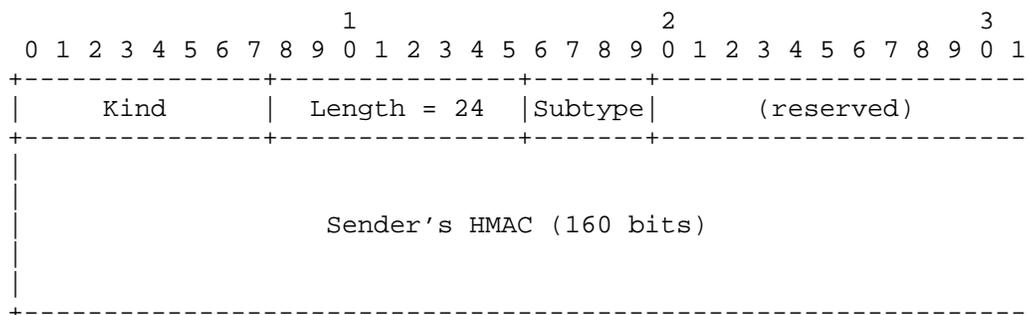
Figure 7: Join Connection (MP_JOIN) option (for third ACK)

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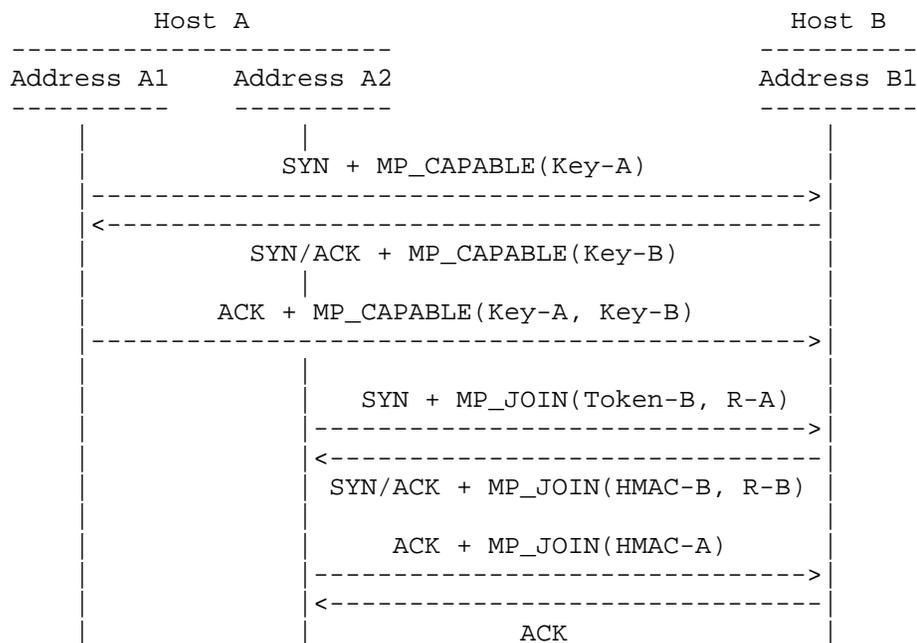
    HMAC-A = HMAC(Key=(Key-A+Key-B), Msg=(R-A+R-B))
    HMAC-B = HMAC(Key=(Key-B+Key-A), Msg=(R-B+R-A))

              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 an 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 setup, demultiplexing
packets is done using the five-tuple, as in traditional TCP. The
five-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 behaviour in detail.

The Data Sequence Mapping and the Data ACK are signalled in the Data
Sequence Signal (DSS) option. Either or both can be signalled 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
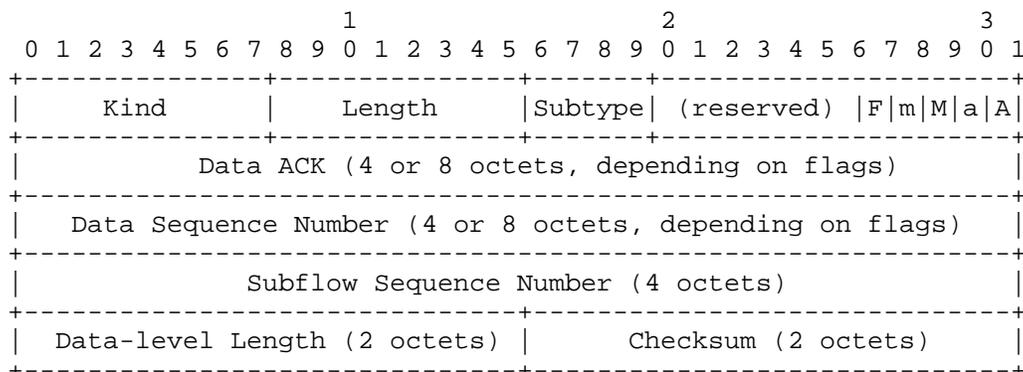signalled in the DSS option, dependent on the flags set.

```
                             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, Subflow Sequence Number, 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, along with 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 [10] 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)      |
   +------------------------------+-------------------------------+
```
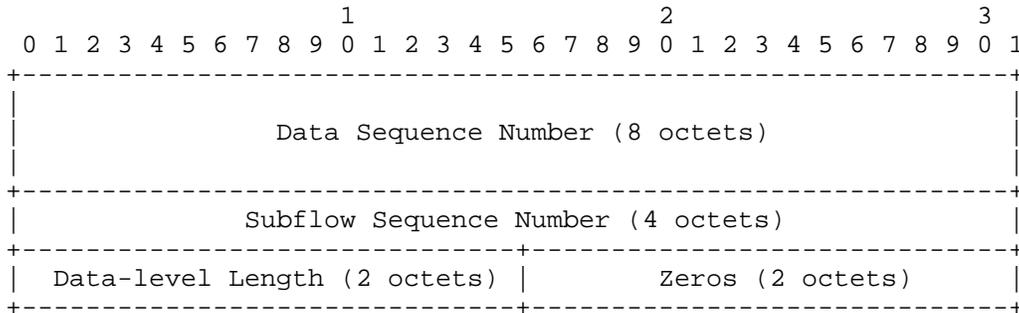
                 Figure 10: Pseudo-Header for DSS Checksum

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, and 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 an 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) [15]. 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 fallback 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 acknowledgement, 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 behaviour 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 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 acknowledgement 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 the data was sent on. 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 optimisations
SHOULD be considered in more advanced implementations, where the Data

ACK is present in segments only when the Data ACK value advances, and
this behaviour MUST be treated as valid. This behaviour 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, and 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 signalled by setting the 'F' flag in the Data
Sequence Signal option (Figure 9) to 1. A DATA_FIN occupies one
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 sequnce space 80-89, the DATA_FIN is DSN 90, and 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 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 behaviour 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 that 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 an 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 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 either put
in the in-order receive queue or in the out-of-order queue. In
multipath TCP, the same happens but at 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 level. 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 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 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 re-send 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 behaviour. Optimisations
could be negotiated in future versions of this protocol.

This protocol specification does not mandate any mechanisms for
handling retransmissions, and much will be dependent upon local
policy (as discussed in Section 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 re-transmission by middleboxes that pro-
actively 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 it
has been sent on. In this way, the sender can always retransmit the
data if needed, on the same subflow or on a different one. A special
case is when a subflow fails: the sender will typically resend the
data on other working subflows 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 an 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 mis-diagnosed 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 maximise 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 ECN signal [16],
   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) |
+--------------+--------------+-------+-----+-+--------------+
```
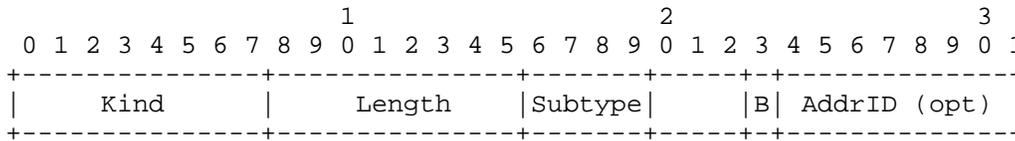
                     Figure 11: MP_PRIO option

   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 signalled 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 separate
   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
   multi-addressed, 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 succesfully setup, 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 which 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 learnt 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 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 two 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)   |
     +---------------------------+
```
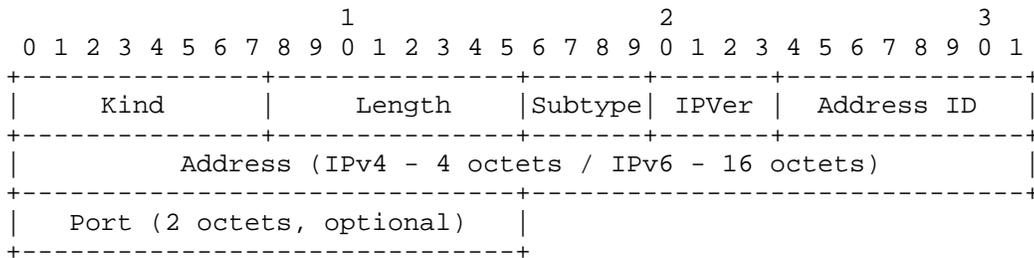
                   Figure 12: Add Address (ADD_ADDR) option

Due to the proliferation of NATs, it is reasonably likely that one

host may attempt to advertise private addresses [17]. 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 minimise
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 mis-direct, 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 minimise 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 minimise 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 to 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 setup 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
[11], 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 [18] 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)
```
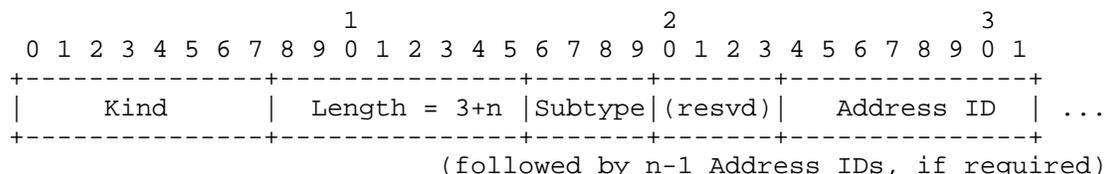
           Figure 13: Remove Address (REMOVE_ADDR) option

3.5. Fast Close

   Regular TCP has the means of sending a reset signal (RST) 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 any more. 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
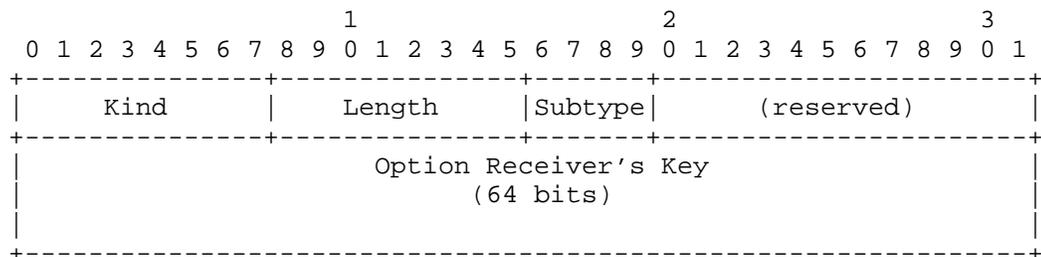
   If Host A wants to force the closure of an MPTCP connection, the
   MPTCP Fast Close procedure is as follows:

   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. Hose 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 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 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 either be 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, behaviour 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 which 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
acknowledgement, 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 an 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 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 an RST.

The case described above is a specialised 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
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 an 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) which 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)               |
   |                                                              |
   +--------------------------------------------------------------+
```
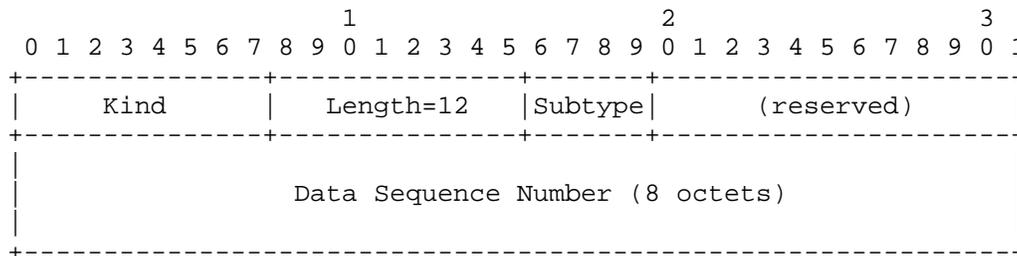
Figure 15: Fallback (MP_FAIL) option

The receiver MUST discard all data following the data sequence number
specified. Failed data MUST NOT be DATA_ACKed and so will be re-
transmitted 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, refering 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 an 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 emphasised 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 an RST - are covered in Section 4. Where possible, we do not want
to deviate from regular TCP behaviour.

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

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
   behaviour 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 which are not required for protocol correctness. In
   this section we detail such heuristics. Note that discussion of
   buffering and certain sender and receiver window behaviours are
   presented in Section 3.3.4 and Section 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 which 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 behaviour.

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. Behaviour
could also be learnt 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 one octet of sequence space. There is an explicit mapping
   of data sequence space to subflow sequence space, which is
   signalled through TCP options in data packets.

ACK: The ACK field in the TCP header acknowledges only the subflow
   sequence number, not the data-level sequence space.
   Implementations SHOULD NOT attempt to infer a data-level
   acknowledgement 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 [8], 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 Section 3.1 and Section 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 whilst 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 (HMAC) 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, and 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 [13]
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 draft should therefore protect against
all forms of flooding and hijacking attacks discussed in [8].

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 indivudal 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 analyis ([8]), 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 signalling 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 [19]), or perhaps
   building on the current approach with a second stage of MPTCP-
   option-based security.

o re-visiting the working group's decision to exclusively use TCP
   options for MPTCP signalling, 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
   behaviour. In this section we outline a few representative
   middlebox-related failure scenarios and show how multipath TCP
   handles them. Next, we list the design decisions multipath has made
   to 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
```
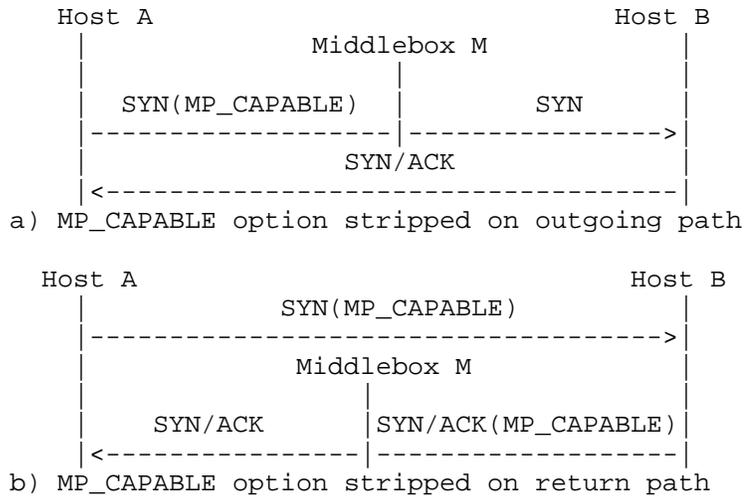
         Figure 16: Connection Setup with Middleboxes that Strip Options from
                                     Packets

   We now examine data flow with MPTCP, assuming the flow is correctly
   setup, which implies the options in the SYN packets were allowed
   through by the relevant middleboxes. If options are allowed through
   and there is no resegmentation or coalescing to TCP segments,
   multipath TCP flows can proceed without problems.

   The case when options get stripped on data packets has been discussed
   in the Fallback section. If a fraction of options are stripped,
   behaviour 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 behaviour
   and the elements in the MPTCP design that allow operation through
   such middleboxes. Issues surrounding dropping packets with options
   or stripping options were discussed above, and are not included here:

   o NATs [20] (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 [17] 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) [21] might pro-actively 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 [22] 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 [23] 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 multi-addressed 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 analyse 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 re-
   assemble for analysis.

o Application level middleboxes such as content-aware firewalls may
   alter the payload within a subflow, such as re-writing 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 fallback
   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 | (This document) |
             +------+--------+--------------+----------------+
```

Table 1: TCP Option Kind Numbers

This document also defines a four-bit subtype field, for which IANA
is to create and maintain a new sub-registry entitled "MPTCP option
subtype values" under the 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 [24].
Assignments consist of the MPTCP subtype's symbolic name and its
associated value, as per the following table.

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

Table 2: MPTCP Option Subtypes

The value 0xf is reserved for Private Use within controlled testbeds.

This document also requests that IANA creates another sub-registry,
"MPTCP handshake algorithms" under the TCP Paramers registry, based
on the flags in MP_CAPABLE (Section 3.1). The flags consist of eight
bits, labelled "A" through "H", and this document assigns the bits as
follows, where "(available)" means that the bit is available for
future assignment:

```
+----------+-------------------+---------------------------+
| Flag Bit |      Meaning      |         Reference         |
+----------+-------------------+---------------------------+
|    A     | Checksum required | This document, Section 3.1 |
|    B     |   Extensibility   | This document, Section 3.1 |
|    C     |    (available)    |                           |
|    D     |    (available)    |                           |
|    E     |    (available)    |                           |
|    F     |    (available)    |                           |
|    G     |    (available)    |                           |
|    H     |     HMAC-SHA1     | This document, 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 [24]. 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", draft-ietf-mptcp-api-05 (work in progress),
          April 2012.

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

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

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

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

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

     [12] Gont, F., "Survey of Security Hardening Methods for
          Transmission Control Protocol (TCP) Implementations",
          draft-ietf-tcpm-tcp-security-03 (work in progress), March 2012.

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

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

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

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

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

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

     [19] Ramaiah, A., "TCP option space extension",
          draft-ananth-tcpm-tcpoptext-00 (work in progress), March 2012.

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

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

[22] 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>.

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

[24] 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 draft.

   SYN packets typically include 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 bi-

directional 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 the port number is present
or not. 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.

   RFC793 [1] specifies several state variables. Whenever possible, we
   reuse the same terminology as RFC793 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 RFC1323, 16 bits without): 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 which 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
   which 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 which is sent in the DSS option on
   all subflows.

RCV.WND (32bits with RFC1323, 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 acknowledgement 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
   which 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 RFC1323, 16 bits otherwise): This is the
    subflow-level receive window which 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
```
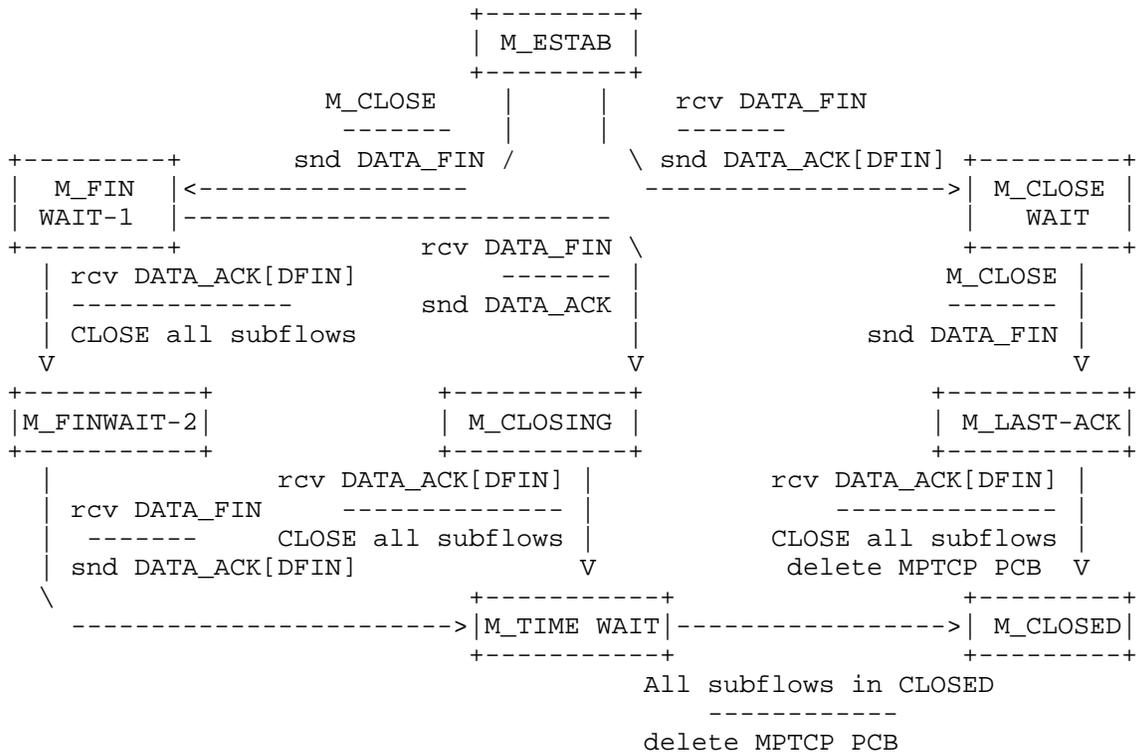
Figure 17: Finite State Machine for Connection Closure

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

            Threat Analysis for TCP Extensions for Multi-path Operation with
                              Multiple Addresses
                         draft-ietf-mptcp-threat-08

Abstract

   Multi-path TCP (MPTCP for short) describes the extensions proposed
   for TCP so that endpoints of a given TCP connection can use multiple
   paths to exchange data. Such extensions enable the exchange of
   segments using different source-destination address pairs, resulting
   in the capability of using multiple paths in a significant number of
   scenarios. Some level of multihoming and mobility support can be
   achieved through these extensions. However, the support for multiple
   IP addresses per endpoint may have implications on the security of
   the resulting MPTCP protocol. This note includes a threat analysis
   for MPTCP.

Status of this Memo

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

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

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

   This Internet-Draft will expire on July 30, 2011.

publication of this document. Please review these documents
carefully, as they describe your rights and restrictions with respect
to this document. Code Components extracted from this document must
include Simplified BSD License text as described in Section 4.e of
the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

Table of Contents

1. Introduction

   Multi-path TCP (MPTCP for short) describes the extensions proposed
   for TCP [RFC0793] so that endpoints of a given TCP connection can use
   multiple paths to exchange data. Such extensions enable the exchange
   of segments using different source-destination address pairs,
   resulting in the capability of using multiple paths in a significant
   number of scenarios. Some level of multihoming and mobility support
   can be achieved through these extensions. However, the support for
   multiple IP addresses per endpoint may have implications on the
   security of the resulting MPTCP protocol. This note includes a
   threat analysis for MPTCP. There are there may other ways to provide
   multiple paths for a TCP connection other than the usage of multiple
   addresses. The threat analysis performed in this document is limited
   to the specific case of using multiple addresses per endpoint.

2. Scope

   There are multiple ways to achieve Multi-path TCP. Essentially what
   is needed is for different segments of the communication to be
   forwarded through different paths by enabling the sender to specify
   some form of path selector. There are multiple options for such a
   path selector, including the usage of different next hops, using
   tunnels to different egress points and so on. In this note, we will
   focus on a particular approach, namely MPTCP, that rely on the usage
   of multiple IP address per endpoint and that uses different source-
   destination address pairs as a mean to express different paths. So,
   in the rest of this note, the MPTCP expression will refer to this
   Multi-addressed flavour of Multi-path TCP
   [I-D.ietf-mptcp-multiaddressed].

   In this note we perform a threat analysis for MPTCP. Introducing the
   support of multiple addresses per endpoint in a single TCP connection
   may result in additional vulnerabilities compared to single-path TCP.
   The scope of this note is to identify and characterize these new
   vulnerabilities. So, the scope of the analysis is limited to the
   additional vulnerabilities resulting from the multi-address support
   compared to the current TCP protocol (where each endpoint only has
   one address available for use per connection). A full analysis of
   the complete set of threats is explicitly out of the scope. The goal
   of this analysis is to help the MPTCP protocol designers create an
   MPTCP specification that is as secure as the current TCP. It is a
   non-goal of this analysis to help in the design of MPTCP that is more
   secure than regular TCP.

   We will focus on attackers that are not along the path, at least not
   during the whole duration of the connection. In the current single

path TCP, an on-path attacker can launch a significant number of
attacks, including eavesdropping, connection hijacking Man-in-the-
Middle attacks and so on. However, it is not possible for the off-
path attackers to launch such attacks. There is a middle ground in
case the attacker is located along the path for a short period of
time to launch the attack and then moves away, but the attack effects
still apply. These are the so-called time-shifted attacks. Since
these are not possible in today's TCP, we will also consider them as
part of the analysis. So, summarizing, we will consider both attacks
launched by off-path attackers and time-shifted attacks. Attacks
launched by on-path attackers are out of scope, since they also apply
to current single-path TCP.

However, that some current on-path attacks may become more difficult
with multi-path TCP, since an attacker (on a single path) will not
have visibility of the complete data stream.

3. Related work

There is a significant amount of previous work in terms of analysis
of protocols that support address agility. In this section we
present the most relevant ones and we relate them to the current
MPTCP effort.

Most of the problems related to address agility have been deeply
analyzed and understood in the context of Route Optimization support
in Mobile IPv6 (MIPv6 RO) [RFC3775]. [RFC4225] includes the rational
for the design of the security of MIPv6 RO. All the attacks
described in the aforementioned analysis apply here and are an
excellent basis for our own analysis. The main differences are:
o In MIPv6 RO, the address binding affects all the communications
   involving an address, while in the MPTCP case, a single connection
   is at stake. If a binding between two addresses is created at the
   IP layer, this binding can and will affect all the connections
   that involve those addresses. However, in MPTCP, if an additional
   address is added to an ongoing TCP connection, the additional
   address will/can only affect the connection at hand and not other
   connections even if the same address is being used for those other
   connections. The result is that in MPTCP there is much less at
   stake and the resulting vulnerabilities are less. On the other
   hand, it is very important to keep the assumption valid that the
   address bindings for a given connection do not affect other
   connections. If reusing of binding or security information is to
   be considered, this assumption could be no longer valid and the
   full impact of the vulnerabilities must be assessed.

   o In MIPv6 there is a trusted third party, called the Home Agent
      that can help with some security problems, as expanded in the next
      bullet.
   o In MIPv6 RO, there is the assumption that the original address
      (Home Address) through which the connection has been established
      is always available and in case it is not, the communication will
      be lost. This is achieved by leveraging in the on the trusted
      party (the Home Agent) to rely the packets to the current location
      of the Mobile Node. In MPTCP, it is an explicit goal to provide
      communication resilience when one of the address pairs is no
      longer usable, so it is not possible to leverage on the original
      address pair to be always working.
   o MIPv6 RO is of course designed for IPv6 and it is an explicit goal
      of MPTCP to support both IPv6 and IPv4. Some MIPv6 RO security
      solutions rely on the usage of some characteristics of IPv6 (such
      as the usage of CGAs [RFC3972]), which will not be usable in the
      context of MPTCP.
   o As opposed to MPTCP, MIPv6 RO does not have a connection state
      information, including sequence numbers, port numbers that could
      be leveraged to provide security in some form.

   In the Shim6 [RFC5533] design, similar issues related to address
   agility were considered and a threat analysis was also performed
   [RFC4218]. The analysis performed for Shim6 also largely applies to
   the MPTCP context, the main difference being:
   o The Shim6 protocol is a layer 3 protocol so all the communications
      involving the target address are at stake; in MPTCP, the impact
      can be limited to a single TCP connection.
   o Similarly to MIPv6 RO, Shim6 only uses IPv6 addresses as
      identifiers and leverages on some of their properties to provide
      the security, such as relying on CGAs or HBAs [RFC5535], which is
      not possible in the MPTCP case where IPv4 addresses must be
      supported.
   o Similarly to MIPv6 RO, Shim6 does not have a connection state
      information, including sequence numbers, port that could be
      leveraged to provide security in some form.

   SCTP [RFC4960]is a transport protocol that supports multiple
   addresses per endpoint and the security implications are very close
   to the ones of MPTCP. A security analysis, identifying a set of
   attacks and proposed solutions was performed in [RFC5062]. The
   results of this analysis apply directly to the case of MPTCP.
   However, the analysis was performed after the base SCTP protocol was
   designed and the goal of the document was essentially to improve the
   security of SCTP. As such, the document is very specific to the
   actual SCTP specification and relies on the SCTP messages and
   behaviour to characterize the issues. While some them can be
   translated to the MPTCP case, some may be caused by specific

behaviour of SCTP.

So, the conclusion is that while we do have a significant amount of previous work that is closely related and we can and will use it as a basis for this analysis, there is a set of characteristics that are specific to MPTCP that grant the need for a specific analysis for MPTCP. The goal of this analysis is to help MPTCP protocol designers to include a set of security mechanisms that prevent the introduction of new vulnerabilities to the Internet due to the adoption of MPTCP.

4. Basic MPTCP.

The goal of this document is to serve as input for MPTCP protocol designers to properly take into account the security issues. As such, the analysis cannot be performed for a specific MPTCP specification, but must be a general analysis that applies to the widest possible set of MPTCP designs. We will characterize what are the fundamental features that any MPTCP protocol must provide and attempt to perform the security implications only assuming those. In some cases, we will have a design choice that will significantly influence the security aspects of the resulting protocol. In that case we will consider both options and try to characterize both designs.

We assume that any MPTCP will behave in the case of a single address per endpoint as TCP. This means that a MPTCP connection will be established by using the TCP 3-way handshake and will use a single address pair.

The addresses used for the establishment of the connection do have a special role in the sense that this is the address used as identifier by the upper layers. The address used as destination address in the SYN packet is the address that the application is using to identify the peer and has been obtained either through the DNS (with or without DNSSEC validation) or passed by a referral or manually introduced by the user. As such, the initiator does have a certain amount of trust in the fact that it is establishing a communication with that particular address. If due to MPTCP, packets end up being delivered to an alternative address, the trust that the initiator has placed on that address would be deceived. In any case, the adoption of MPTCP necessitates a slight evolution of the traditional TCP trust model, in that the initiator is additionally trusting the peer to provide additional addresses which it will trust to the same degree as the original pair. An application or implementation that cannot trust the peer in this way should not make use of multiple paths.

During the 3-way handshake, the sequence number will be synchronized

for both ends, as in regular TCP. We assume that a MPTCP connection
will use a single sequence number for the data, even if the data is
exchanged through different paths, as MPTCP provides an in-order
delivery service of bytes

Once the connection is established, the MPTCP extensions can be used
to add addresses for each of the endpoints. This is achieved by each
end sending a control message containing the additional address(es).
In order to associate the additional address to an ongoing
connection, the connection needs to be identified. We assume that
the connection can be identified by the 4-tuple of source address,
source port, destination address, destination port used for the
establishment of the connection. So, at least, the control message
that will convey the additional address information can also contain
the 4-tuple in order to inform about what connection the address
belong to (if no other connection identifier is defined). There are
two different ways to convey address information:
o Explicit mode: the control message contain a list of addresses.
o Implicit mode: the address added is the one included in the source
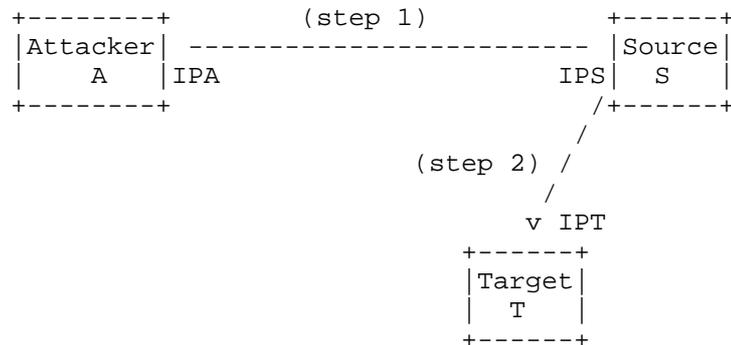   address field of the IP header

These two modes have different security properties for some type of
attacks. The explicit mode seems to be the more vulnerable to abuse.
The implicit mode may benefit from forms of ingress filtering
security, which would reduce the possibility of an attacker to add
any arbitrary address to an ongoing connection. However, ingress
filtering deployment is far from universal, and it is unwise to rely
on it as a basis for the protection of MPTCP.

Further consideration about the interaction between ingress filtering
and implicit mode signaling is needed in the case that we need to
remove an address that is no longer available from the MPTCP
connection. A host attached to a network that performs ingress
filtering and using implicit signaling would not be able to remove an
address that is no longer available (either because of a failure or
due to a mobility event) from an ongoing MPTCP connection.

We will assume that MPTCP will use all the address pairs that it has
available for sending packets and that it will distribute the load
based on congestion among the different paths.

5. Flooding attacks

The first type of attacks that are introduced by address agility are
the flooding (or bombing) attacks. The setup for this attack is
depicted in the following figure:

```
         +--------+        (step 1)              +------+
         |Attacker| ------------------------ |Source|
         |   A    |IPA                      IPS|  S   |
         +--------+                            /+------+
                                             /
                                 (step 2) /
                                         /
                                    v IPT
                              +------+
                              |Target|
                              |  T   |
                              +------+
```

The scenario consists of an attacker A who has an IP address IPA. A
server that can generate a significant amount of traffic (such as a
streaming server), called source S and that has IP address IPS.
Target T has an IP address IPT.

In step 1 of this attack, the attacker A establishes a MPTCP
connection with the source of the traffic server S and starts
downloading a significant amount of traffic. The initial connection
only involves one IP address per endpoint, IPA and IPS. Once that
the download is on course, in the step 2 of the attack is that the
attacker A adds IPT as one of the available addresses for the
communication. How the additional address is added depends on the
MPTCP address management mode. In explicit address management, the
attacker A only needs to send a signaling packet conveying address
IPT. In implicit mode, the attacker A would need to send a packet
with IPT as the source address. Depending on whether ingress
filtering is deployed and the location of the attacker, it may be
possible or not for the attacker to send such a packet. At this
stage, the MPTCP connection still has a single address for the Source
S i.e. IPS but has two addresses for the Attacker A, IPA and IPT.
The attacker now attempts to get the Source S to send the traffic of
the ongoing download to the Target T IP address i.e. IPT. The
attacker can do that by pretending that the path between IPA and IPT
is congested but that the path between IPS and IPT is not. In order
to do that, it needs to send ACKs for the data that flows through the
path between IPS and IPT and do not send ACKs for the data that is
sent to IPA. The details of this will depend on how the data sent
through the different paths is ACKed. One possibility is that ACKs
for the data sent using a given address pair should come in packets
containing the same address pair. If so, the attacker would need to
send ACKs using packets containing IPT as the source address to keep
the attack flowing. This may be possible or not depending on the
deployment of ingress filtering and the location of the attacker.
The attacker would also need to guess the sequence number of the data

being sent to the Target. Once the attacker manages to perform these
actions the attack is on place and the download will hit the target.
In this type of attacks, the Source S still thinks it is sending
packets to the Attacker A while in reality it is sending the packet
to Target T.

Once that the traffic from the Source S start hitting the Target T,
the target will react. Since the packets are likely to belong to a
non existent TCP connection, the Target T will issue RST packets. It
is relevant then to understand how MPTCP reacts to incoming RST
packets. It seems that the at least the MPTCP that receives a RST
packet should terminate the packet exchange corresponding to the
particular address pair (maybe not the complete MPTCP connection, but
at least it should not send more packets with the address pair
involved in the RST packet). However, if the attacker, before
redirecting the traffic has managed to increase the window size
considerably, the flight size could be enough to impose a significant
amount of traffic to the Target node. There is a subtle operation
that the attacker needs to achieve in order to launch a significant
attack. On the one hand it needs to grow the window enough so that
the flight size is big enough to cause enough effect and on the other
hand the attacker needs to be able to simulate congestion on the IPA-
IPS path so that traffic is actually redirected to the alternative
path without significantly reducing the window. This will heavily
depend on how the coupling of the windows between the different paths
works, in particular how the windows are increased. Some designs of
the congestion control window coupling could render this attack
ineffective. If the MPTCP protocol requires performing slow start
per subflow, then the flooding will be limited by the slow-start
initial window size.

Previous protocols, such as MIPv6 RO and SCTP, that have to deal with
this type of attacks have done so by adding a reachability check
before actually sending data to a new address. The solution used in
other protocols, would include the Source S to explicitly asking the
host sitting in the new address (the Target T sitting in IPT) whether
it is willing to accept packets from the MPTCP connection identified
by the 4-tuple IPA, port A, IPS, port S. Since this is not part of
the established connection that Target T has, T would not accept the
request and Source S would not use IPT to send packets for this MPTCP
connection. Usually, the request also includes a nonce that cannot
be guessed by the attacker A so that it cannot fake the reply to the
request easily. In the case of SCTP, it sends a message with a 64-
bit nonce (in a HEARTBEAT).

One possible approach to do this reachability test would be to
perform a 3-way handshake for each new address pair that is going to
be used in a MPTCP connection. While there are other reasons for

   doing this (such as NAT traversal), such approach would also act as a
   reachability test and would prevent the flooding attacks described in
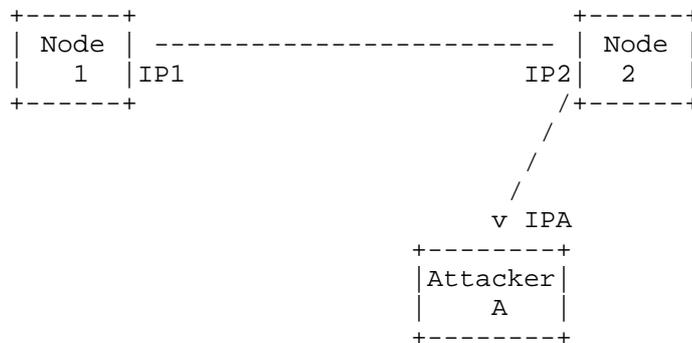   this section.

   Another type of flooding attack that could potentially be performed
   with MPTCP is one where the attacker initiates a communication with a
   peer and includes a long list of alternative addresses in explicit
   mode. If the peer decides to establish subflows with all the
   available addresses, the attacker have managed to achieve an
   amplified attack, since by sending a single packet containing all the
   alternative addresses it triggers the peer to generate packets to all
   the destinations.

6. Hijacking attacks

6.1. Hijacking attacks to the Basic MPTCP protocol

   The hijacking attacks essentially use the MPTCP address agility to
   allow an attacker to hijack a connection. This means that the victim
   of a connection thinks that it is talking to a peer, while it is
   actually exchanging packets with the attacker. In some sense it is
   the dual of the flooding attacks (where the victim thinks it is
   exchanging packets with the attacker but in reality is sending the
   packets to the target).

   The scenario for a hijacking attack is described in the next figure.


```
          +------+                           +------+
          | Node | -----------------------   | Node |
          |  1   |IP1                     IP2|  2   |
          +------+                          /+------+
                                           /
                                          /
                                         /
                                        v IPA
                                 +--------+
                                 |Attacker|
                                 |   A    |
                                 +--------+
```
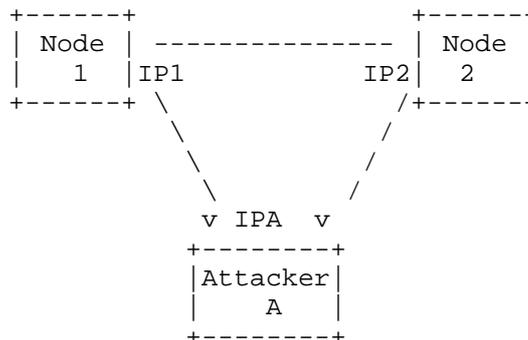
   We have a MPTCP connection established between Node 1 and Node 2.
   The connection is using only one address per endpoint, IP1 and IP2.
   The attacker then launches the hijacking attack by adding IPA as an
   additional address for Node 1. There is not much difference between

explicit or implicit address management, since in both cases the
Attacker A could easily send a control packet adding the address IPA,
either as control data or as the source address of the control
packet. In order to be able to hijack the connection, the attacker
needs to know the 4-tuple that identifies the connection, including
the pair of addresses and the pair of ports. It seems reasonable to
assume that knowing the source and destination IP addresses and the
port of the server side is fairly easy for the attacker. Learning
the port of the client (i.e. of the initiator of the connection) may
prove to be more challenging. The attacker would need to guess what
the port is or to learn it by intercepting the packets. Assuming
that the attacker can gather the 4-tuple and issue the message adding
IPA to the addresses available for the MPTCP connection, then the
attacker A has been able to participate in the communication. In
particular:

o Segments flowing from the Node 2:Depending how the usage of
   addresses is defined, Node 2 will start using IPA to send data to.
   In general, since the main goal is to achieve multi-path
   capabilities, we can assume that unless there are already many IP
   address pairs in use in the MPTCP connection, Node 2 will start
   sending data to IPA. This means that part of the data of the
   communication will reach the Attacker but probably not all of it.
   This already has negative effects, since Node 1 will not receive
   all the data from Node 2. Moreover, from the application
   perspective, this would result in DoS attack, since the byte flow
   will stop waiting for the missing data. However, it is not enough
   to achieve full hijacking of the connection, since part of data
   will be still delivered to IP1, so it would reach Node 1 and not
   the Attacker. In order for the attacker to receive all the data
   of the MPTCP connection, the Attacker must somehow remove IP1 of
   the set of available addresses for the connection. in the case of
   implicit address management, this operation is likely to imply
   sending a termination packet with IP1 as source address, which may
   or not be possible for the attacker depending on whether ingress
   filtering is in place and the location of the attacker. If
   explicit address management is used, then the attacker will send a
   remove address control packet containing IP1. Once IP1 is
   removed, all the data sent by Node 2 will reach the Attacker and
   the incoming traffic has been hijacked.

o Segments flowing to the Node 2: As soon as IPA is accepted by Node
   2 as part of the address set for the MPTCP connection, the
   Attacker can send packets using IPA and those packets will be
   considered by Node 2 as part of MPTCP connection. This means that
   the attacker will be able to inject data into the MPTCP
   connection, so from this perspective, the attacker has hijacked
   part of the outgoing traffic. However, Node 1 would still be able
   to send traffic that will be received by Node 2 as part of the
   MPTCP connection. This means that there will be two sources of

data i.e. Node 1 and the attacker, potentially preventing the
full hijacking of the outgoing traffic by the attacker. In order
to achieve a full hijacking, the attacker would need to remove IP1
from the set of available addresses. This can be done using the
same techniques described in the previous paragraph.

A related attack that can be achieved using similar techniques would
be a Man-in-the-Middle (MitM) attack. The scenario for the attack is
depicted in the figure below.

```
        +------+                  +------+
        | Node | --------------- | Node |
        |   1  |IP1           IP2|  2   |
        +------+ \              /+------+
                  \            /
                   \          /
                    \        /
                     v IPA  v
                  +--------+
                  |Attacker|
                  |   A    |
                  +--------+
```

There is an established connection between Node 1 and Node 2. The
Attacker A will use the MPTCP address agility capabilities to place
itself as a MitM. In order to do so, it will add IP address IPA as
an additional address for the MPTCP connection on both Node 1 and
Node 2. This is essentially the same technique described earlier in
this section, only that it is used against both nodes involved in the
communication. The main difference is that in this case, the
attacker can simply sniff the content of the communication that is
forwarded through it and in turn forward the data to the peer of the
communication. The result is that the attacker can place himself in
the middle of the communication and sniff part of the traffic
unnoticed. Similar considerations about how the attacker can manage
to get to see all the traffic by removing the genuine address of the
peer apply.

6.2. Time-shifted hijacking attacks

A simple way to prevent off-path attackers to launch hijacking
attacks is to provide security of the control messages that add and
remove addresses by the usage of a cookie. In this type of
approaches, the peers involved in the MPTCP connection agree on a
cookie, that is exchanged in plain text during the establishment of

the connection and that needs to be presented in every control packet
that adds or removes an address for any of the peers. The result is
that the attacker needs to know the cookie in order to launch any of
the hijacking attacks described earlier. This implies that off path
attackers can no longer perform the hijacking attacks and that only
on-path attackers can do so, so one may consider that a cookie based
approach to secure MPTCP connection results in similar security than
current TCP. While it is close, it is not entirely true.

The main difference between the security of a MPTCP protocol secured
through cookies and the current TCP protocol are the time shifted
attacks. As we described earlier, a time shifted attack is one where
the attacker is along the path during a period of time, and then
moves away but the effects of the attack still remains, after the
attacker is long gone. In the case of a MPTCP protocol secured
through the usage of cookies, the attacker needs to be along the path
until the cookie is exchanged. After the attacker has learnt the
cookie, it can move away from the path and can still launch the
hijacking attacks described in the previous section.

There are several types of approaches that provide some protection
against hijacking attacks and that are vulnerable to some forms of
time-shifted attacks. We will next present some general taxonomy of
solutions and we describe the residual threats:
o Cookie-based solution: As we described earlier, one possible
   approach is to use a cookie, that is sent in clear text in every
   MPTCP control message that adds a new address to the existing
   connection. The residual threat in this type of solution is that
   any attacker that can sniff any of these control messages will
   learn the cookie and will be able to add new addresses at any
   given point in the lifetime of the connection. Moreover, the
   endpoints will not detect the attack since the original cookie is
   being used by the attacker. Summarizing, the vulnerability window
   of this type of attacks includes all the flow establishment
   exchanges and it is undetectable by the endpoints.
o Shared secret exchanged in plain text: An alternative option that
   is more secure than the cookie based approach is to exchange a key
   in clear text during the establishment of the first subflow and
   then validate the following subflows by using a keyed HMAC
   signature using the shared key. This solution would be vulnerable
   to attackers sniffing the message exchange for the establishment
   of the first subflow, but after that, the shared key is not
   transmitted any more, so the attacker cannot learn it through
   sniffing any other message. Unfortunately, in order to be
   compatible with NATs (see analysis below) even though this
   approach includes a keyed HMAC signature, this signature cannot
   cover the IP address that is being added. This means that this
   type of approaches are also vulnerable to integrity attacks of the

exchanged messages. This means that even though the attacker
cannot learn the shared key by sniffing the subsequent subflow
establishment, the attacker can modify the subflow establishment
message and change the address that is being added. So, the
vulnerability window for confidentially to the shared key is
limited to the establishment of the first subflow, but the
vulnerability window for integrity attacks still includes all the
subflow establishment exchanges. These attacks are still
undetectable by the endpoints. The SCTP security falls in this
category.
o Strong crypto anchor exchange. Another approach that could be
used would be to exchange some strong crypto anchor while the
establishment of the first subflow, such as a public key or a hash
chain anchor. Subsequent subflows could be protected by using the
crypto material associated to that anchor. An attacker in this
case would need to change the crypto material exchanged in the
connection establishment phase. As a result the vulnerability
window for forging the crypto anchor is limited to the initial
connection establishment exchange. Similarly to the previous
case, due to NAT traversal considerations, the vulnerability
window for integrity attacks include all the subflow establishment
exchanges. Because the attacker needs to change the crypto
anchor, this approach are detectable by the endpoints, if they
communicate directly.

6.3. NAT considerations

   In order to be widely adopted MPTCP must work through NATs. NATs are
   an interesting device from a security perspective. In terms of MPTCP
   they essentially behave as a Man-in-the-Middle attacker. MPTCP
   security goal is to prevent from any attacker to insert their
   addresses as valid addresses for a given MPTCP connection. But that
   is exactly what a NAT does, they modify the addresses. So, if MPTCP
   is to work through NATs, MPTCP must accept address rewritten by NATs
   as valid addresses for a given session. The most direct corollary is
   that the MPTCP messages that add addresses in the implicit mode (i.e.
   the SYN of new subflows) cannot be protected against integrity
   attacks, since they must allow for NATs to change their addresses.
   This rules out any solution that would rely on providing integrity
   protection to prevent an attacker from changing the address used in a
   subflow establishment exchange. This implies that alternative
   creative mechanisms are needed to protect from integrity attacks to
   the MPTCP signaling that adds new addresses to a connection. It is
   far from obvious how one such creative approach could look like at
   this point.

   In the case of explicit mode, you could protect the address included
   in the MPTCP option. Now the question is what address to include in

the MPTCP option that conveys address information. If the address
included is the address configured in the host interface and that
interface is behind a NAT, the address information is useless, as the
address is not actually reachable from the other end so there is no
point in conveying it and even less in securing it. It would be
possible to envision the usage of NAT traversal techniques such as
STUN to learn the address and port that the NAT has assigned and
convey that information in a secure. While this is possible, it
relies on using NAT traversal techniques and also tools to convey the
address and the port in a secure manner.

7. Recommendation

   The presented analysis shows that there is a tradeoff between the
   complexity of the security solution and the residual threats. In
   order to define a proper security solution, we need to assess the
   tradeoff and make a recommendation. After evaluating the different
   aspects in the MPTCP WG, our conclusion are as follows:

   MPTCP should implement some form of reachability check using a random
   nonce (e.g. TCP 3-way handshake) before adding a new address to an
   ongoing communication in order to prevent flooding attacks.

   The default security mechanisms for MPTCP should be to exchange a key
   in clear text in the establishment of the first subflow and then
   secure following address additions by using a keyed HMAC using the
   exchanged key.

   MPTCP security mechanism should support using a pre-shared key to be
   used in the keyed HMAC, providing a higher level of protection than
   the previous one.

   A mechanism to prevent replay attacks using these messages should be
   provided e.g. a sequence number protected by the HMAC.

   The MPTCP protocol should be extensible and it should able to
   accommodate multiple security solutions, in order to enable the usage
   of more secure mechanisms if needed.

8. Security Considerations

   This note contains a security analysis for MPTCP, so no further
   security considerations need to be described in this section.

9. IANA Considerations

   This document does not require any action from IANA.

10. Contributors

   Alan Ford - Roke Manor Research Ltd.

11. Acknowledgments

   Rolf Winter, Randall Stewart, Andrew McDonald, Michael Tuexen,
   Michael Scharf, Tim Shepard, Yoshifumi Nishida, Lars Eggert, Phil
   Eardley, Jari Arkko, David Harrington, Dan Romascanu, Alexey Melnikov
   reviewed an earlier version of this document and provided comments to
   improve it.

   Mark Handley pointed out the problem with NATs and integrity
   protection of MPTCP signaling.

   Marcelo Bagnulo is partly funded by Trilogy, a research project
   supported by the European Commission under its Seventh Framework
   Program.

12. References

12.1. Normative References

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

12.2. Informative References

   [RFC4225] Nikander, P., Arkko, J., Aura, T., Montenegro, G., and E.
              Nordmark, "Mobile IP Version 6 Route Optimization Security
              Design Background", RFC 4225, December 2005.

   [RFC4218] Nordmark, E. and T. Li, "Threats Relating to IPv6
              Multihoming Solutions", RFC 4218, October 2005.

   [RFC3972] Aura, T., "Cryptographically Generated Addresses (CGA)",
              RFC 3972, March 2005.

   [RFC5062] Stewart, R., Tuexen, M., and G. Camarillo, "Security
              Attacks Found Against the Stream Control Transmission
              Protocol (SCTP) and Current Countermeasures", RFC 5062,

                   September 2007.

   [RFC5535] Bagnulo, M., "Hash-Based Addresses (HBA)", RFC 5535,
              June 2009.

   [RFC3775] Johnson, D., Perkins, C., and J. Arkko, "Mobility Support
              in IPv6", RFC 3775, June 2004.

   [RFC5533] Nordmark, E. and M. Bagnulo, "Shim6: Level 3 Multihoming
              Shim Protocol for IPv6", RFC 5533, June 2009.

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

   [I-D.ietf-mptcp-multiaddressed]
              Ford, A., Raiciu, C., and M. Handley, "TCP Extensions for
              Multipath Operation with Multiple Addresses",
              draft-ietf-mptcp-multiaddressed-02 (work in progress),
              October 2010.

Author's Address

   Marcelo Bagnulo
   Universidad Carlos III de Madrid
   Av. Universidad 30
   Leganes, Madrid 28911
   SPAIN

   Phone: 34 91 6248814
   Email: marcelo@it.uc3m.es
   URI: http://www.it.uc3m.es

This Internet-Draft, draft-raiciu-mptcp-congestion-01.txt, has been replaced by another
document, draft-ietf-mptcp-congestion-00.txt, and has been deleted from the
Internet-Drafts directory.

Internet-Drafts are not archival documents, and copies of Internet-Drafts
that have been deleted from the directory are not available. The
Secretariat does not have any information regarding the future plans of
the author(s) or working group, if applicable, with respect to this
deleted Internet-Draft. For more information, or to request a copy of
the document, please contact the author(s) directly.

Draft Author(s):

 M. Handley <m.handley@cs.ucl.ac.uk>
 C. Raiciu <c.raiciu@cs.ucl.ac.uk>
 D. Wischik <d.wischik@cs.ucl.ac.uk>