

Internet Engineering Task Force  
Internet-Draft  
Intended status: Informational  
Expires: October 19, 2012

M. Scharf  
Alcatel-Lucent Bell Labs  
A. Ford  
Cisco  
April 17, 2012

MPTCP Application Interface Considerations  
draft-ietf-mptcp-api-05

Abstract

Multipath TCP (MPTCP) adds the capability of using multiple paths to a regular TCP session. Even though it is designed to be totally backward compatible to applications, the data transport differs compared to regular TCP, and there are several additional degrees of freedom that applications may wish to exploit. This document summarizes the impact that MPTCP may have on applications, such as changes in performance. Furthermore, it discusses compatibility issues of MPTCP in combination with non-MPTCP-aware applications. Finally, the document describes a basic application interface which is a simple extension of TCP's interface for MPTCP-aware applications.

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 October 19, 2012.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents

(<http://trustee.ietf.org/license-info>) in effect on the date of 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 . . . . .	4
2. Terminology . . . . .	5
3. Comparison of MPTCP and Regular TCP . . . . .	5
3.1. Performance Impact . . . . .	5
3.1.1. Throughput . . . . .	6
3.1.2. Delay . . . . .	6
3.1.3. Resilience . . . . .	7
3.2. Potential Problems . . . . .	7
3.2.1. Impact of Middleboxes . . . . .	7
3.2.2. Impact on Implicit Assumptions . . . . .	8
3.2.3. Security Implications . . . . .	8
4. Operation of MPTCP with Legacy Applications . . . . .	9
4.1. Overview of the MPTCP Network Stack . . . . .	9
4.2. Address Issues . . . . .	10
4.2.1. Specification of Addresses by Applications . . . . .	10
4.2.2. Querying of Addresses by Applications . . . . .	10
4.3. MPTCP Connection Management . . . . .	11
4.3.1. Reaction to Close Call by Application . . . . .	11
4.3.2. Other Connection Management Functions . . . . .	11
4.4. Socket Option Issues . . . . .	12
4.4.1. General Guideline . . . . .	12
4.4.2. Disabling of the Nagle Algorithm . . . . .	12
4.4.3. Buffer Sizing . . . . .	12
4.4.4. Other Socket Options . . . . .	13
4.5. Default Enabling of MPTCP . . . . .	13
4.6. Summary of Advices to Application Developers . . . . .	13
5. Basic API for MPTCP-aware Applications . . . . .	14
5.1. Design Considerations . . . . .	14
5.2. Requirements on the Basic MPTCP API . . . . .	14
5.3. Sockets Interface Extensions by the Basic MPTCP API . . . . .	16
5.3.1. Overview . . . . .	16
5.3.2. Enabling and Disabling of MPTCP . . . . .	17
5.3.3. Binding MPTCP to Specified Addresses . . . . .	18
5.3.4. Querying the MPTCP Subflow Addresses . . . . .	18
5.3.5. Getting a Unique Connection Identifier . . . . .	19
6. Other Compatibility Issues . . . . .	19
6.1. Usage of the SCTP Socket API . . . . .	19
6.2. Incompatibilities with other Multihoming Solutions . . . . .	19

6.3. Interactions with DNS . . . . .	20
7. Security Considerations . . . . .	20
8. IANA Considerations . . . . .	21
9. Conclusion . . . . .	21
10. Acknowledgments . . . . .	21
11. References . . . . .	22
11.1. Normative References . . . . .	22
11.2. Informative References . . . . .	22
Appendix A. Requirements on a Future Advanced MPTCP API . . . . .	23
A.1. Design Considerations . . . . .	23
A.2. MPTCP Usage Scenarios and Application Requirements . . . . .	24
A.3. Potential Requirements on an Advanced MPTCP API . . . . .	26
A.4. Integration with the SCTP Socket API . . . . .	27
Appendix B. Change History of the Document . . . . .	28

## 1. Introduction

Multipath TCP adds the capability of using multiple paths to a regular TCP session [1]. The motivations for this extension include increasing throughput, overall resource utilisation, and resilience to network failure, and these motivations are discussed, along with high-level design decisions, as part of the Multipath TCP architecture [4]. The MPTCP protocol [5] offers the same reliable, in-order, byte-stream transport as TCP, and is designed to be backward compatible with both applications and the network layer. It requires support inside the network stack of both endpoints.

This document first presents the impacts that MPTCP may have on applications, such as performance changes compared to regular TCP. Second, it defines the interoperation of MPTCP and applications that are unaware of the multipath transport. MPTCP is designed to be usable without any application changes, but some compatibility issues have to be taken into account. Third, this memo specifies a basic Application Programming Interface (API) for MPTCP-aware applications. The API presented here is an extension to the regular TCP API to allow an MPTCP-aware application the equivalent level of control and access to information of an MPTCP connection that would be possible with the standard TCP API on a regular TCP connection.

The de facto standard API for TCP/IP applications is the "sockets" interface. This document provides an abstract definition of MPTCP-specific extensions to this interface. These are operations that can be used by an application to get or set additional MPTCP-specific information on a socket, in order to provide an equivalent level of information and control over MPTCP as exists for an application using regular TCP. It is up to the applications, high-level programming languages, or libraries to decide whether to use these optional extensions. For instance, an application may want to turn on or off the MPTCP mechanism for certain data transfers, or limit its use to certain interfaces. The abstract specification is in line with the Posix standard [17] as much as possible.

An advanced API for MPTCP is outside the scope of this document. Such an advanced API could offer a more fine-grained control over multipath transport functions and policies. The appendix includes a brief, non-compulsory list of potential features of such an advanced API.

There can be interactions or incompatibilities of MPTCP with other APIs or socket interface extensions, which are discussed later in this document. Some network stack implementations, specially on mobile devices, have centralized connection managers or other higher-level APIs to solve multi-interface issues, as surveyed in [15].

Their interaction with MPTCP is outside the scope of this note.

The target readers of this document are application developers whose software may benefit significantly from MPTCP. This document also provides the necessary information for developers of MPTCP to implement the API in a TCP/IP network stack.

## 2. Terminology

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

This document uses the MPTCP terminology introduced in [5].

Concerning the API towards applications, the following terms are distinguished:

- o Legacy API: The interface towards TCP that is currently used by applications. This document explains the impact of MPTCP for such applications, as well as resulting issues.
- o Basic API: A simple extension of TCP's interface for applications that are aware of MPTCP. This document abstractly describes this interface, which provides access to multipath address information and a level of control equivalent to regular TCP.
- o Advanced API: An API that offers more fine-grained control over the MPTCP behavior. Its specification is outside scope of this document.

## 3. Comparison of MPTCP and Regular TCP

This section discusses the impact that the use of MPTCP will have on applications, in comparison to what may be expected from the use of regular TCP.

### 3.1. Performance Impact

One of the key goals of adding multipath capability to TCP is to improve the performance of a transport connection by load distribution over separate subflows across potentially disjoint paths. Furthermore, it is an explicit goal of MPTCP that it should not provide a worse performing connection than would have existed through the use of single-path TCP. A corresponding congestion control algorithm is described in [7]. The following sections summarize the performance impact of MPTCP as seen by an application.

### 3.1.1. Throughput

The most obvious performance improvement that will be gained with the use of MPTCP is an increase in throughput, since MPTCP will pool more than one path (where available) between two endpoints. This will provide greater bandwidth for an application. If there are shared bottlenecks between the flows, then the congestion control algorithms will ensure that load is evenly spread amongst regular and multipath TCP sessions, so that no end user receives worse performance than if all were using single-path TCP.

This performance increase additionally means that an MPTCP session could achieve throughput that is greater than the capacity of a single interface on the device. If any applications make assumptions about interfaces due to throughput (or vice versa), they must take this into account (although an MPTCP implementation must always respect an application's request for a particular interface).

Furthermore, the flexibility of MPTCP to add and remove subflows as paths change availability could lead to a greater variation, and more frequent change, in connection bandwidth. Applications that adapt to available bandwidth (such as video and audio streaming) may need to adjust some of their assumptions to most effectively take this into account.

The transport of MPTCP signalling information results in a small overhead. If multiple subflows share a same bottleneck, this overhead slightly reduces the capacity that is available for data transport. Yet, this potential reduction of throughput will be negligible in many usage scenarios, and the protocol contains optimisations in its design so that this overhead is minimal.

### 3.1.2. Delay

If the delays on the constituent subflows of an MPTCP connection differ, the jitter perceivable to an application may appear higher as the data is spread across the subflows. Although MPTCP will ensure in-order delivery to the application, the application must be able to cope with the data delivery being burstier than may be usual with single-path TCP. Since burstiness is commonplace on the Internet today, it is unlikely that applications will suffer from such an impact on the traffic profile, but application authors may wish to consider this in future development.

In addition, applications that make round trip time (RTT) estimates at the application level may have some issues. Whilst the average delay calculated will be accurate, whether this is useful for an application will depend on what it requires this information for. If

a new application wishes to derive such information, it should consider how multiple subflows may affect its measurements, and thus how it may wish to respond. In such a case, an application may wish to express its scheduling preferences, as described later in this document.

### 3.1.3. Resilience

Another performance improvement through the use of MPTCP is better resilience. The use of multiple subflows simultaneously means that, if one should fail, all traffic will move to the remaining subflow(s), and additionally any lost packets can be retransmitted on these subflows.

As one special case, the MPTCP protocol can be used with only one active subflow at a given point in time. In that case, resilience compared to single-path TCP is improved, too. MPTCP also supports make-before-break and break-before-make handovers between subflows. In both cases, the MPTCP connection can survive an unavailability or change of an IP address (e.g., due to shutdown of an interface or handover). MPTCP close or resets the MPTCP connection separately from the individual subflows, as described in [5].

Subflow failure may be caused by issues within the network, which an application would be unaware of, or interface failure on the node. An application may, under certain circumstances, be in a position to be aware of such failure (e.g. by radio signal strength, or simply an interface enabled flag), and so must not make assumptions of an MPTCP flow's stability based on this. An MPTCP implementation must never override an application's request for a given interface, however, so the cases where this issue may be applicable are limited.

## 3.2. Potential Problems

### 3.2.1. Impact of Middleboxes

MPTCP has been designed in order to pass through the majority of middleboxes. Empirical evidence suggests that new TCP options can successfully be used on most paths in the Internet [18]. Nevertheless some middleboxes may still refuse to pass MPTCP messages due to the presence of TCP options, or they may strip TCP options. If this is the case, MPTCP falls back to regular TCP. Although this will not create a problem for the application (its communication will be set up either way), there may be additional (and indeed, user-perceivable) delay while the first handshake fails. Therefore, an alternative approach could be to try both MPTCP and regular TCP connection attempts at the same time, and respond to whichever replies first (or apply a timeout on the MPTCP attempt, while having

TCP SYN/ACK ready to reply to, thus reducing the setup delay by a RTT) in a similar fashion to the "Happy Eyeballs" mechanism for IPv6 [16].

An MPTCP implementation can learn the rate of MPTCP connection attempt successes or failures to particular hosts or networks, and on particular interfaces, and could therefore learn heuristics of when and when not to use MPTCP. A detailed discussion of the various fallback mechanisms, for failures occurring at different points in the connection, is presented in [5].

There may also be middleboxes that transparently change the length of content. If such middleboxes are present, MPTCP's reassembly of the byte stream in the receiver is difficult. Still, MPTCP can detect such middleboxes and then fall back to regular TCP. An overview of the impact of middleboxes is presented in [4] and MPTCP's mechanisms to work around these are presented and discussed in [5].

MPTCP can also have other unexpected implications. For instance, intrusion detection systems could be triggered. A full analysis of MPTCP's impact on such middleboxes is for further study after deployment experiments.

### 3.2.2. Impact on Implicit Assumptions

In regular TCP, there is a one-to-one mapping of the socket interface to a flow through a network. Since MPTCP can make use of multiple subflows, applications cannot implicitly rely on this one-to-one mapping any more. Whilst this doesn't matter for most applications, a few applications require the transport along a single path; they can disable the use of MPTCP as described later in this document. Examples include monitoring tools that want to measure the available bandwidth on a path, or routing protocols such as BGP that require the use of a specific link.

Furthermore, an implementation may choose to persist an MPTCP connection even if an IP address is not allocated any more to a host, depending on the policy concerning the first subflow (fate-sharing, see Section 4.2.2). In this case, the IP address exposed to an MPTCP-unaware application can differ to the addresses actually being used by MPTCP. It is even possible that an IP address gets assigned to another host during the lifetime of an MPTCP connection.

### 3.2.3. Security Implications

The support for multiple IP addresses within one MPTCP connection can result in additional security vulnerabilities, such as possibilities for attackers to hijack connections. The protocol design of MPTCP



minimizes this risk. An attacker on one of the paths can cause harm, but this is hardly an additional security risk compared to single-path TCP, which is vulnerable to man-in-the-middle attacks, too. A detailed threat analysis of MPTCP is published in [6].

#### 4. Operation of MPTCP with Legacy Applications

##### 4.1. Overview of the MPTCP Network Stack

MPTCP is an extension of TCP, but it is designed to be backward compatible for legacy (MPTCP-unaware) applications. TCP interacts with other parts of the network stack by different interfaces. The de facto standard API between TCP and applications is the sockets interface. The position of MPTCP in the protocol stack is illustrated in Figure 1.

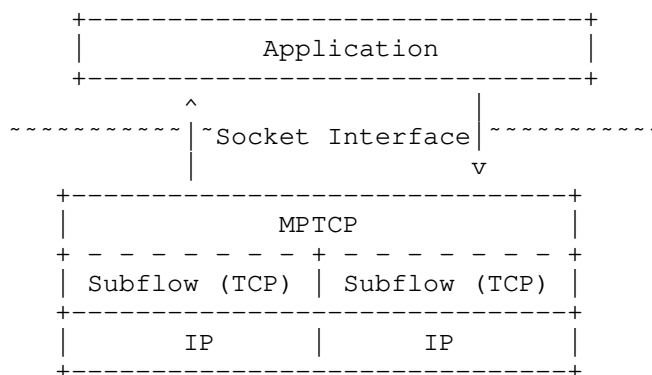


Figure 1: MPTCP protocol stack

In general, MPTCP can affect all interfaces that make assumptions about the coupling of a TCP connection to a single IP address and TCP port pair, to one sockets endpoint, to one network interface, or to a given path through the network.

This means that there are two classes of applications:

- o Legacy applications: These applications are unaware of MPTCP and use the existing API towards TCP without any changes. This is the default case.
- o MPTCP-aware applications: These applications indicate support for an enhanced MPTCP interface. This document specifies a minimum set of API extensions for such applications.

In the following, it is discussed to what extent MPTCP affects legacy

applications using the existing sockets API. The existing sockets API implies that applications deal with data structures that store, amongst others, the IP addresses and TCP port numbers of a TCP connection. A design objective of MPTCP is that legacy applications can continue to use the established sockets API without any changes. However, in MPTCP there is a one-to-many mapping between the socket endpoint and the subflows. This has several subtle implications for legacy applications using sockets API functions.

## 4.2. Address Issues

### 4.2.1. Specification of Addresses by Applications

During binding, an application can either select a specific address, or bind to `INADDR_ANY`. Furthermore, on some systems other socket options (e.g., `SO_BINDTODEVICE`) can be used to bind to a specific interface. If an application uses a specific address or binds to a specific interface, then MPTCP MUST respect this and not interfere in the application's choices. The binding to a specific address or interface implies that the application is not aware of MPTCP and will disable the use of MPTCP on this connection. An application that wishes to bind to a specific set of addresses with MPTCP must use multipath-aware calls to achieve this (as described in Section 5.3.3).

If an application binds to `INADDR_ANY`, it is assumed that the application does not care which addresses to use locally. In this case, a local policy MAY allow MPTCP to automatically set up multiple subflows on such a connection.

The basic sockets API of MPTCP-aware applications allows to express further preferences in an MPTCP-compatible way (e.g. bind to a subset of interfaces only).

### 4.2.2. Querying of Addresses by Applications

Applications can use the `getpeername()` or `getsockname()` functions in order to retrieve the IP address of the peer or of the local socket. These functions can be used for various purposes, including security mechanisms, geo-location, or interface checks. The socket API was designed with an assumption that a socket is using just one address, and since this address is visible to the application, the application may assume that the information provided by the functions is the same during the lifetime of a connection. However, in MPTCP, unlike in TCP, there is a one-to-many mapping of a connection to subflows, and subflows can be added and removed while the connections continues to exist. Since the subflow addresses can change, MPTCP cannot expose addresses by `getpeername()` or `getsockname()` that are both valid and

constant during the connection's lifetime.

This problem is addressed as follows: If used by a legacy application, the MPTCP stack MUST always return the addresses of the first subflow of an MPTCP connection, in all circumstances, even if that particular subflow is no longer in use.

As this address may not be valid any more if the first subflow is closed, the MPTCP stack MAY close the whole MPTCP connection if the first subflow is closed (i.e. fate sharing between the initial subflow and the MPTCP connection as a whole). Whether to close the whole MPTCP connection by default SHOULD be controlled by a local policy. Further experiments are needed to investigate its implications.

The functions `getpeername()` and `getsockname()` SHOULD also always return the addresses of the first subflow if the socket is used by an MPTCP-aware application, in order to be consistent with MPTCP-unaware applications, and, e. g., also with SCTP. Instead of `getpeername()` or `getsockname()`, MPTCP-aware applications can use new API calls, documented later, in order to retrieve the full list of address pairs for the subflows in use.

#### 4.3. MPTCP Connection Management

##### 4.3.1. Reaction to Close Call by Application

As described in [5], MPTCP distinguishes between the closing of subflows (by TCP FIN) and closing the whole MPTCP connection (by DATA FIN).

When an application closes a socket, e.g., by calling the `close()` function, this indicates that the application has no more data to send, like for single-path TCP. MPTCP will then close the MPTCP connection by DATA FIN messages. This is completely transparent for an application.

In summary, the semantics of the `close()` interface for applications are not changed compared to TCP.

##### 4.3.2. Other Connection Management Functions

In general, an MPTCP connection is maintained separately from individual subflows. The MPTCP protocol therefore has internal mechanisms to establish, close, or reset the MPTCP connection [5]. They provide equivalent functions like single-path TCP and can be mapped accordingly. Therefore, these MPTCP internals do not affect the application interface.

#### 4.4. Socket Option Issues

##### 4.4.1. General Guideline

The existing sockets API includes options that modify the behavior of sockets and their underlying communications protocols. Various socket options exist on socket, TCP, and IP level. The value of an option can usually be set by the `setsockopt()` system function. The `getsockopt()` function gets information. In general, the existing sockets interface functions cannot configure each MPTCP subflow individually. In order to be backward compatible, existing APIs therefore **SHOULD** apply to all subflows within one connection, as far as possible.

##### 4.4.2. Disabling of the Nagle Algorithm

One commonly used TCP socket option (`TCP_NODELAY`) disables the Nagle algorithm as described in [2]. This option is also specified in the Posix standard [17]. Applications can use this option in combination with MPTCP exactly in the same way. It then **SHOULD** disable the Nagle algorithm for the MPTCP connection, i.e., all subflows.

In addition, the MPTCP protocol instance **MAY** use a different path scheduler algorithm if `TCP_NODELAY` is present. For instance, it could use an algorithm that is optimized for latency-sensitive traffic (for instance only transmitting on one path). Specific algorithms are outside the scope of this document.

##### 4.4.3. Buffer Sizing

Applications can explicitly configure send and receive buffer sizes by the sockets API (`SO_SNDBUF`, `SO_RCVBUF`). These socket options can also be used in combination with MPTCP and then affect the buffer size of the MPTCP connection. However, when defining buffer sizes, application programmers should take into account that the transport over several subflows requires a certain amount of buffer for resequencing in the receiver. MPTCP may also require more storage space in the sender, in particular, if retransmissions are sent over more than one path. In addition, very small send buffers may prevent MPTCP from efficiently scheduling data over different subflows. Therefore, it does not make sense to use MPTCP in combination with small send or receive buffers.

An MPTCP implementation **MAY** set a lower bound for send and receive buffers and treat a small buffer size request as an implicit request not to use MPTCP.

#### 4.4.4. Other Socket Options

Some network stacks also provide other implementation-specific socket options or interfaces that affect TCP's behavior. If a network stack supports MPTCP, it must be ensured that these options do not interfere.

#### 4.5. Default Enabling of MPTCP

It is up to a local policy at the end system whether a network stack should automatically enable MPTCP for sockets even if there is no explicit sign of MPTCP awareness of the corresponding application. Such a choice may be under the control of the user through system preferences.

The enabling of MPTCP, either by application or by system defaults, does not necessarily mean that MPTCP will always be used. Both endpoints must support MPTCP, and there must be multiple addresses at at least one endpoint, for MPTCP to be used. Even if those requirements are met, however, MPTCP may not be immediately used on a connection. It may make sense for multiple paths to be brought into operation only after a given period of time, or if the connection is saturated.

#### 4.6. Summary of Advices to Application Developers

- o Using the default MPTCP configuration: Like TCP, MPTCP is designed to be efficient and robust in the default configuration. Application developers should not explicitly configure TCP (or MPTCP) features unless this is really needed.
- o Socket buffet dimensioning: Multipath transport requires larger buffers in the receiver for resequencing, as already explained. Applications should use reasonable buffer sizes (such as the operating system default values) in order to fully benefit from MPTCP. A full discussion of buffer sizing issues is given in [5].
- o Facilitating stack-internal heuristics: The path management and data scheduling by MPTCP is realized by stack-internal algorithms that may implicitly try to self-optimize their behavior according to assumed application needs. For instance, an MPTCP implementation may use heuristics to determine whether an application requires delay-sensitive or bulk data transport, using for instance port numbers, the TCP\_NODELAY socket options, or the application's read/write patterns as input parameters. An application developer can facilitate the operation of such heuristics by avoiding atypical interface use cases. For instance, for long bulk data transfers, it does neither make sense

to enable the TCP\_NODELAY socket option, nor is it reasonable to use many small socket "send()" calls each with small amounts of data only.

## 5. Basic API for MPTCP-aware Applications

### 5.1. Design Considerations

While applications can use MPTCP with the unmodified sockets API, multipath transport results in many degrees of freedom. MPTCP manages the data transport over different subflows automatically. By default, this is transparent to the application, but an application could use an additional API to interface with the MPTCP layer and to control important aspects of the MPTCP implementation's behavior.

This document describes a basic MPTCP API. The API contains a minimum set of functions that provide an equivalent level of control and information as exists for regular TCP. It maintains backward compatibility with legacy applications.

An advanced MPTCP API is outside the scope of this document. The basic API does not allow a sender or a receiver to express preferences about the management of paths or the scheduling of data, even if this can have a significant performance impact and if an MPTCP implementation could benefit from additional guidance by applications. A list of potential further API extensions is provided in the appendix. The specification of such an advanced API is for further study and may partly be implementation-specific.

MPTCP mainly affects the sending of data. But a receiver may also have preferences about data transfer choices, and it may have performance requirements, too. A receiver may also have preferences about data transfer choices, and it may have performance requirements, too. Yet, the configuration of such preferences is outside of the scope of the basic API.

### 5.2. Requirements on the Basic MPTCP API

Because of the importance of the sockets interface there are several fundamental design objectives for the basic interface between MPTCP and applications:

- o Consistency with existing sockets APIs must be maintained as far as possible. In order to support the large base of applications using the original API, a legacy application must be able to continue to use standard socket interface functions when run on a system supporting MPTCP. Also, MPTCP-aware applications should be able to access the socket without any major changes.

- o Sockets API extensions must be minimized and independent of an implementation.
- o The interface should handle both IPv4 and IPv6.

The following is a list of the core requirements for the basic API:

- REQ1: Turn on/off MPTCP: An application should be able to request to turn on or turn off the usage of MPTCP. This means that an application should be able to explicitly request the use of MPTCP if this is possible. Applications should also be able to request not to enable MPTCP and to use regular TCP transport instead. This can be implicit in many cases, since MPTCP must be disabled by the use of binding to a specific address. MPTCP may also be enabled if an application uses a dedicated multipath address family (such as AF\_MULTIPATH, [8]).
- REQ2: An application should be able to restrict MPTCP to binding to a given set of addresses.
- REQ3: An application should be able to obtain information on the pairs of addresses used by the MPTCP subflows.
- REQ4: An application should be able to extract a unique identifier for the connection (per endpoint).

The first requirement is the most important one, since some applications could benefit a lot from MPTCP, but there are also cases in which it hardly makes sense. The existing sockets API provides similar mechanisms to enable or disable advanced TCP features. The second requirement corresponds to the binding of addresses with the `bind()` socket call, or, e.g., explicit device bindings with a `SO_BINDTODEVICE` option. The third requirement ensures that there is an equivalent to `getpeername()` or `getsockname()` that is able to deal with more than one subflow. Finally, it should be possible for the application to retrieve a unique connection identifier (local to the endpoint on which it is running) for the MPTCP connection. This replaces the (address, port) pair for a connection identifier in single-path TCP, which is no longer static in MPTCP.

An application can continue to use `getpeername()` or `getsockname()` in addition to the basic MPTCP API. Both functions return the corresponding addresses of the first subflow, as already explained.

### 5.3. Sockets Interface Extensions by the Basic MPTCP API

#### 5.3.1. Overview

The abstract, basic MPTCP API consists of a set of new values that are associated with an MPTCP socket. Such values may be used for changing properties of an MPTCP connection, or retrieving information. These values could be accessed by new symbols on existing calls such as `setsockopt()` and `getsockopt()`, or could be implemented as entirely new function calls. This implementation decision is out of scope for this document. The following list presents symbolic names for these MPTCP socket settings.

- o `TCP_MULTIPATH_ENABLE`: Enable/disable MPTCP
- o `TCP_MULTIPATH_ADD`: Bind MPTCP to a set of given local addresses, or add a new local address to an existing MPTCP connection
- o `TCP_MULTIPATH_REMOVE`: Remove a local address from an MPTCP connection
- o `TCP_MULTIPATH_SUBFLOWS`: Get the pairs of addresses currently used by the MPTCP subflows
- o `TCP_MULTIPATH_CONNID`: Get the local connection identifier for this MPTCP connection

Table 1 shows a list of the abstract socket operations for the basic configuration of MPTCP. The first column gives the symbolic name of the operation. The second and third columns indicate whether the operation provides values to be read ("Get") or takes values to configure ("Set"). The fourth column lists the type of data associated with this operation.

Name	Get	Set	Data type
<code>TCP_MULTIPATH_ENABLE</code>	o	o	boolean
<code>TCP_MULTIPATH_ADD</code>		o	list of addresses
<code>TCP_MULTIPATH_REMOVE</code>		o	list of addresses
<code>TCP_MULTIPATH_SUBFLOWS</code>	o		list of pairs of addresses
<code>TCP_MULTIPATH_CONNID</code>	o		32-bit integer

Table 1: MPTCP Socket Operations

There are restrictions when these new socket operations can be used:



- o `TCP_MULTIPATH_ENABLE`: This value SHOULD only be set before the establishment of a TCP connection. Its value SHOULD only be read after the establishment of a connection.
- o `TCP_MULTIPATH_ADD`: This operation can be both applied before connection setup or during a connection. If used before, it controls the local addresses that an MPTCP connection can use. In the latter case, it allows MPTCP to use an additional local address, if there has been a restriction before connection setup.
- o `TCP_MULTIPATH_REMOVE`: This operation can be both applied before connection setup or during a connection. In both cases, it removes an address from the list of local addresses that may be used by subflows.
- o `TCP_MULTIPATH_SUBFLOWS`: This value is read-only and SHOULD only be used after connection setup.
- o `TCP_MULTIPATH_CONNID`: This value is read-only and SHOULD only be used after connection setup.

#### 5.3.2. Enabling and Disabling of MPTCP

An application can explicitly indicate multipath capability by setting `TCP_MULTIPATH_ENABLE` to a value larger than 0. In this case, the MPTCP implementation SHOULD try to negotiate MPTCP for that connection. Note that multipath transport will not necessarily be enabled, as it requires support at both end systems, no middleboxes on the path that would prevent any additional signalling, and at least one endpoint with multiple addresses.

Building on the backwards-compatibility specified in Section 4.2.1, if an application enables MPTCP but binds to a specific address or interface, MPTCP MUST be enabled, but MPTCP MUST respect the application's choice and only use addresses that are explicitly provided by the application. Note that it would be possible for an application to use the legacy bindings, and then expand on them by using `TCP_MULTIPATH_ADD`. Note also that it is possible for more than one local address to be initially available to MPTCP in this case, if an application has bound to a specific interface with multiple addresses.

An application can disable MPTCP setting `TCP_MULTIPATH_ENABLE` to a value of 0. In that case, MPTCP MUST NOT be used on that connection.

After connection establishment, an application can get the value of `TCP_MULTIPATH_ENABLE`. A value of 0 then means lack of MPTCP support. Any value equal to or larger than 1 means that MPTCP is supported.

### 5.3.3. Binding MPTCP to Specified Addresses

Before connection establishment, an application can use `TCP_MULTIPATH_ADD` function to indicate a set of local IP addresses that MPTCP may bind to. The parameter of the function is a list of addresses in a corresponding data structure. By extension, this operation will also control the list of addresses that can be advertised to the peer via MPTCP signalling.

If an application binds to a specific address or interface, it is not required to use the `TCP_MULTIPATH_ADD` operation for that address. As explained in Section 5.3.2, MPTCP MUST only use the explicitly specified addresses in that case.

An application MAY also indicate a TCP port number that, if specified, MPTCP MUST attempt to bind to. The port number MAY be different to the one used by existing subflows. If no port number is provided by the application, the port number is automatically selected by the MPTCP implementation, and will usually be the same across all subflows.

This operation can also be used to modify the address list in use during the lifetime of an MPTCP connection. In this case, it is used to indicate a set of additional local addresses that the MPTCP connection can make use of, and which can be signalled to the peer. It should be noted that this signal is only a hint, and an MPTCP implementation MAY only use a subset of the addresses.

The `TCP_MULTIPATH_REMOVE` operation can be used to remove a (set of) local addresses from an MPTCP connection. MPTCP MUST close any corresponding subflows (i.e. those using the local address that is no longer present), and signal the removal of the address to the peer. If alternative paths are available using the supplied address list but MPTCP is not currently using them, an MPTCP implementation SHOULD establish alternative subflows before undertaking the address removal.

It should be remembered that these operations SHOULD support both IPv4 and IPv6 addresses, potentially in the same call.

### 5.3.4. Querying the MPTCP Subflow Addresses

An application can get a list of the addresses used by the currently established subflows in an MPTCP connection by means of the read-only `TCP_MULTIPATH_SUBFLOWS` operation.

The return value is a list of pairs of tuples of IP address and TCP port number. In one pair, the first tuple refers to the local IP

address and the local TCP port, and the second one to the remote IP address and remote TCP port used by the subflow. The list MUST only include established subflows. Both addresses in each pair MUST be either IPv4 or IPv6.

#### 5.3.5. Getting a Unique Connection Identifier

An application that wants a unique identifier for the connection, analogous to an (address, port) pair in regular TCP, can query the TCP\_MULTIPATH\_CONNID value to get a local connection identifier for the MPTCP connection.

This SHOULD be a 32-bit number, and SHOULD be the locally unique (e.g., the MPTCP token).

### 6. Other Compatibility Issues

#### 6.1. Usage of the SCTP Socket API

For dealing with multi-homing, several socket API extensions have been defined for SCTP [13]. As MPTCP realizes multipath transport from and to multi-homed endsystems, some of these interface function calls are actually applicable to MPTCP in a similar way.

API developers MAY wish to integrate SCTP and MPTCP calls to provide a consistent interface to the application. Yet, it must be emphasized that the transport service provided by MPTCP is different to SCTP, and this is why not all SCTP API functions can be mapped directly to MPTCP. Furthermore, a network stack implementing MPTCP does not necessarily support SCTP and its specific socket interface extensions. This is why the basic API of MPTCP defines additional socket options only, which are a backward compatible extension of TCP's application interface. An integration with the SCTP API is outside the scope of the basic API.

#### 6.2. Incompatibilities with other Multihoming Solutions

The use of MPTCP can interact with various related sockets API extensions. The use of a multihoming shim layer conflicts with multipath transport such as MPTCP or SCTP [11]. Care should be taken for the usage not to confuse with the overlapping features of other APIs:

- o SHIM API [11]: This API specifies sockets API extensions for the multihoming shim layer.
- o HIP API [12]: The Host Identity Protocol (HIP) also results in a new API.

- o API for Mobile IPv6 [10]: For Mobile IPv6, a significantly extended socket API exists as well (in addition to API extensions for IPv6 [9]).

In order to avoid any conflict, multiaddressed MPTCP SHOULD NOT be enabled if a network stack uses SHIM6, HIP, or Mobile IPv6. Furthermore, applications should not try to use both the MPTCP API and another multihoming or mobility layer API.

It is possible, however, that some of the MPTCP functionality, such as congestion control, could be used in a SHIM6 or HIP environment. Such operation is for further study.

### 6.3. Interactions with DNS

In multihomed or multiaddressed environments, there are various issues that are not specific to MPTCP, but have to be considered, too. These problems are summarized in [14].

Specifically, there can be interactions with DNS. Whilst it is expected that an application will iterate over the list of addresses returned from a call such as `getaddrinfo()`, MPTCP itself MUST NOT make any assumptions about multiple A or AAAA records from the same DNS query referring to the same host, as it is possible that multiple addresses refer to multiple servers for load balancing purposes.

## 7. Security Considerations

This document first defines the behavior of the standard TCP/IP API for MPTCP-unaware applications. In general, enabling MPTCP has some security implications for applications, which are introduced in Section 5.3.3, and these threats are further detailed in [6]. The protocol specification of MPTCP [5] defines several mechanisms to protect MPTCP against those attacks.

In addition, the basic MPTCP API for MPTCP-aware applications defines functions that provide an equivalent level of control and information as exists for regular TCP. New functions enable adding and removing local addresses from an MPTCP connection (`TCP_MULTIPATH_ADD` and `TCP_MULTIPATH_REMOVE`). These functions don't add security threats if the MPTCP stack verifies that the addresses provided by the application are indeed available as source addresses for subflows.

However, applications should use the `TCP_MULTIPATH_ADD` function with care, as new subflows might get established to those addresses. Furthermore, it could result in some form of information leakage since MPTCP might advertise those addresses to the other connection endpoint, which could learn IP addresses of interfaces that are not

visible otherwise.

Use of different addresses should not be assumed to lead to use of different paths, especially for security purposes.

MPTCP-aware applications should also take care when querying and using information about the addresses used by subflows (TCP\_MULTIPATH\_SUBFLOWS). As MPTCP can dynamically open and close subflows, a list of addresses queried once can get outdated during the lifetime of an MPTCP connection. Then, the list may contain invalid entries, i.e. addresses that are not used any more, or that might not even be assigned to that host any more. Applications that want to ensure that MPTCP only uses a certain set of addresses should explicitly bind to those addresses.

## 8. IANA Considerations

This document has no IANA actions. This document only defines an abstract API and therefore does not request the reservation of identifiers or names.

## 9. Conclusion

This document discusses MPTCP's implications and its performance impact on applications. In addition, it specifies a basic MPTCP API. For legacy applications, it is ensured that the existing sockets API continues to work. MPTCP-aware applications can use the basic MPTCP API that provides some control over the transport layer equivalent to regular TCP.

## 10. Acknowledgments

Authors sincerely thank to the following people for their helpful comments and reviews of the document: Philip Eardley, Lavkesh Lahngir, John Leslie, Costin Raiciu, Michael Tuexen, and Javier Ubillos.

Michael Scharf is supported by the German-Lab project (<http://www.german-lab.de/>) funded by the German Federal Ministry of Education and Research (BMBF). Alan Ford was previously supported by Roke Manor Research and 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.

## 11. References

## 11.1. Normative References

- [1] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.
- [2] Braden, R., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, October 1989.
- [3] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [4] Ford, A., Raiciu, C., Handley, M., Barre, S., and J. Iyengar, "Architectural Guidelines for Multipath TCP Development", RFC 6182, March 2011.
- [5] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", draft-ietf-mptcp-multiaddressed-07 (work in progress), March 2012.
- [6] Bagnulo, M., "Threat Analysis for TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6181, March 2011.
- [7] Raiciu, C., Handley, M., and D. Wischik, "Coupled Congestion Control for Multipath Transport Protocols", RFC 6356, October 2011.

## 11.2. Informative References

- [8] Sarolahti, P., "Multi-address Interface in the Socket API", draft-sarolahti-mptcp-af-multipath-01 (work in progress), March 2010.
- [9] Stevens, W., Thomas, M., Nordmark, E., and T. Jinmei, "Advanced Sockets Application Program Interface (API) for IPv6", RFC 3542, May 2003.
- [10] Chakrabarti, S. and E. Nordmark, "Extension to Sockets API for Mobile IPv6", RFC 4584, July 2006.
- [11] Komu, M., Bagnulo, M., Slavov, K., and S. Sugimoto, "Sockets Application Program Interface (API) for Multihoming Shim", RFC 6316, July 2011.
- [12] Komu, M. and T. Henderson, "Basic Socket Interface Extensions for the Host Identity Protocol (HIP)", RFC 6317, July 2011.
- [13] Stewart, R., Tuexen, M., Poon, K., Lei, P., and V. Yasevich,

- "Sockets API Extensions for the Stream Control Transmission Protocol (SCTP)", RFC 6458, December 2011.
- [14] Blanchet, M. and P. Seite, "Multiple Interfaces and Provisioning Domains Problem Statement", RFC 6418, November 2011.
- [15] Wasserman, M. and P. Seite, "Current Practices for Multiple-Interface Hosts", RFC 6419, November 2011.
- [16] Wing, D. and A. Yourtchenko, "Happy Eyeballs: Success with Dual-Stack Hosts", RFC 6555, April 2012.
- [17] "IEEE Std. 1003.1-2008 Standard for Information Technology -- Portable Operating System Interface (POSIX). Open Group Technical Standard: Base Specifications, Issue 7, 2008."
- [18] Honda, M., Nishida, Y., Raiciu, C., Greenhalgh, A., Handley, M., and H. Tokuda, "Is it Still Possible to Extend TCP?", November 2011.

## Appendix A. Requirements on a Future Advanced MPTCP API

### A.1. Design Considerations

Multipath transport results in many degrees of freedom. The basic MPTCP API only defines a minimum set of the API extensions for the interface between the MPTCP layer and applications, which does not offer much control of the MPTCP implementation's behavior. A future, advanced API could address further features of MPTCP and provide more control.

Applications that use TCP may have different requirements on the transport layer. While developers have become used to the characteristics of regular TCP, new opportunities created by MPTCP could allow the service provided to be optimised further. An advanced API could enable MPTCP-aware applications to specify preferences and control certain aspects of the behavior, in addition to the simple control provided by the basic interface. An advanced API could also address aspects that are completely out-of-scope of the basic API, for example, the question whether a receiving application could influence the sending policy.

Furthermore, an advanced MPTCP API could be part of a new overall interface between the network stack and applications that addresses other issues as well, such as the split between identifiers and locators. An API that does not use IP addresses (but, instead e.g. a `connectbyname()` function) would be useful for numerous purposes,

independent of MPTCP.

It has also been suggested to use a separate address family called AF\_MULTIPATH [8]. This separate address family could be used to exchange multiple addresses between an application and the standard sockets API, but it would be a more fundamental change compared to the basic API described in this document.

This appendix documents a list of potential usage scenarios and requirements for the advanced API. The specification and implementation of a corresponding API is outside the scope of this document.

## A.2. MPTCP Usage Scenarios and Application Requirements

There are different MPTCP usage scenarios. An application that wishes to transmit bulk data will want MPTCP to provide a high throughput service immediately, through creating and maximising utilisation of all available subflows. This is the default MPTCP use case.

But at the other extreme, there are applications that are highly interactive, but require only a small amount of throughput, and these are optimally served by low latency and jitter stability. In such a situation, it would be preferable for the traffic to use only the lowest latency subflow (assuming it has sufficient capacity), maybe with one or two additional subflows for resilience and recovery purposes. The key challenge for such a strategy is that the delay on a path may fluctuate significantly and that just always selecting the path with the smallest delay might result in instability.

The choice between bulk data transport and latency-sensitive transport affects the scheduler in terms of whether traffic should be, by default, sent on one subflow or across several ones. Even if the total bandwidth required is less than that available on an individual path, it is desirable to spread this load to reduce stress on potential bottlenecks, and this is why this method should be the default for bulk data transport. However, that may not be optimal for applications that require latency/jitter stability.

In the case of the latter option, a further question arises: Should additional subflows be used whenever the primary subflow is overloaded, or only when the primary path fails (hot-standby)? In other words, is latency stability or bandwidth more important to the application? This results in two different options: Firstly, there is the single path which can overflow into an additional subflow; and secondly there is single-path with hot-standby, whereby an application may want an alternative backup subflow in order to



improve resilience. In case that data delivery on the first subflow fails, the data transport could immediately be continued on the second subflow, which is idle otherwise.

Yet another complication is introduced with the potential that MPTCP introduces for changes in available bandwidth as the number of available subflows changes. Such jitter in bandwidth may prove confusing for some applications such as video or audio streaming that dynamically adapt codecs based on available bandwidth. Such applications may prefer MPTCP to attempt to provide a consistent bandwidth as far as is possible, and avoid maximising the use of all subflows.

A further, mostly orthogonal question is whether data should be duplicated over the different subflows, in particular if there is spare capacity. This could improve both the timeliness and reliability of data delivery.

In summary, there are at least three possible performance objectives for multipath transport (not necessarily disjoint):

1. High bandwidth
2. Low latency and jitter stability
3. High reliability

In an advanced API, applications could provide high-level guidance to the MPTCP implementation concerning these performance requirements, for instance, which is considered to be the most important one. The MPTCP stack would then use internal mechanisms to fulfill this abstract indication of a desired service, as far as possible. This would both affect the assignment of data (including retransmissions) to existing subflows (e.g., 'use all in parallel', 'use as overflow', 'hot standby', 'duplicate traffic') as well as the decisions when to set up additional subflows to which addresses. In both cases different policies can exist, which can be expected to be implementation-specific.

Therefore, an advanced API could provide a mechanism how applications can specify their high-level requirements in an implementation-independent way. One possibility would be to select one "application profile" out of a number of choices that characterize typical applications. Yet, as applications today do not have to inform TCP about their communication requirements, it requires further studies whether such an approach would be realistic.

Of course, independent of an advanced API, such functionality could

also partly be achieved by MPTCP-internal heuristics that infer some application preferences e.g. from existing socket options, such as TCP\_NODELAY. Whether this would be reliable, and indeed appropriate, is for further study, too.

#### A.3. Potential Requirements on an Advanced MPTCP API

The following is a list of potential requirements for an advanced MPTCP API beyond the features of the basic API. It is included here for information only:

- REQ5: An application should be able to establish MPTCP connections without using IP addresses as locators.
- REQ6: An application should be able obtain usage information and statistics about all subflows (e.g., ratio of traffic sent via this subflow).
- REQ7: An application should be able to request a change in the number of subflows in use, thus triggering removal or addition of subflows. An even finer control granularity would be a request for the establishment of a specific subflow to a provided destination, or a request for the termination of a specified, existing subflow.
- REQ8: An application should be able to inform the MPTCP implementation about its high-level performance requirements, e.g., in the form of a profile.
- REQ9: An application should be able to indicate communication characteristics, e. g., the expected amount of data to be sent, the expected duration of the connection, or the expected rate at which data is provided. Applications may in some cases be able to forecast such properties. If so, such information could be an additional input parameter for heuristics inside the MPTCP implementation, which could be useful for example to decide when to set up additional subflows.
- REQ10: An application should be able to control the automatic establishment/termination of subflows. This would imply a selection among different heuristics of the path manager, e.g., 'try as soon as possible', 'wait until there is a bunch of data', etc.

- REQ11: An application should be able to set preferred subflows or subflow usage policies. This would result in a selection among different configurations of the multipath scheduler. For instance, an application might want to use certain subflows as backup only.
- REQ12: An application should be able to control the level of redundancy by telling whether segments should be sent on more than one path in parallel.

An advanced API fulfilling these requirements would allow application developers to more specifically configure MPTCP. It could avoid suboptimal decisions of internal, implicit heuristics. However, it is unclear whether all of these requirements would have a significant benefit to applications, since they are going above and beyond what the existing API to regular TCP provides.

A subset of this functions might also be implemented system wide or by other configuration mechanisms. These implementation details are left for further study.

#### A.4. Integration with the SCTP Socket API

The advanced API may also integrate or use the SCTP Socket API. The following functions that are defined for SCTP have a similar functionality like the basic MPTCP API:

- o `sctp_bindx()`
- o `sctp_connectx()`
- o `sctp_getladdrs()`
- o `sctp_getpaddrs()`
- o `sctp_freeladdrs()`
- o `sctp_freepaddrs()`

The syntax and semantics of these functions are described in [13].

A potential objective for the advanced API is to provide a consistent MPTCP and SCTP interface to the application. This is left for further study.

## Appendix B. Change History of the Document

Note to RFC Editor: Remove this section before publication

Changes compared to version draft-ietf-mptcp-api-04:

- o Slightly changed abstract (comment by Philip Eardley)
- o Removal of redundant text from intro (comment by Philip Eardley)
- o New text on the lack of interface differences to regular TCP regarding closing the connection, also in the resilience discussion (comment by Philip Eardley)
- o Moved AF\_MULTIPATH to appendix (comment by Philip Eardley)
- o Update of text on connection identifier to align with latest protocol specification (comment by Lavkesh Lahngir)
- o Numerous small editorial changes

Changes compared to version draft-ietf-mptcp-api-03:

- o Security consideration section
- o Better explanation of the implications of explicitly specified addresses, most notably during the bind call
- o Editorial changes

Changes compared to version draft-ietf-mptcp-api-02:

- o Updated references
- o Editorial changes

Changes compared to version draft-ietf-mptcp-api-01:

- o Additional text on outdated assumptions if an MPTCP application does not use fate sharing.
- o The appendix explicitly mentions an integration of the advanced MPTCP API and the SCTP API as a potential objective, which is left for further study for the basic API.
- o A short additional explanation of the parameters of the abstract functions TCP\_MULTIPATH\_ADD and TCP\_MULTIPATH\_REMOVE.

- o Better explanation when TCP\_MULTIPATH\_REMOVE may be used.

Changes compared to version draft-ietf-mptcp-api-00:

- o Explicitly specify that the TCP\_MULTIPATH\_SUBFLOWS function returns port numbers, too. Furthermore, add a new comment that TCP\_MULTIPATH\_ADD permits the specification of a port number.
- o Mention possible additional extended API functions for the indication of application characteristics and for backup paths, based on comments received from the community.
- o Mentions alternative approaches for avoiding non-MPTCP-capable paths to reduce impact on applications.

Changes compared to version draft-scharf-mptcp-api-03:

- o Removal of explicit references to "socket options" and getsockopt/setsockopt.
- o Change of TCP\_MULTIPATH\_BIND to TCP\_MULTIPATH\_ADD and TCP\_MULTIPATH\_REMOVE.
- o Mention of stability of bandwidth as another potential QoS parameter for the advanced API.
- o Address comments received from Philip Eardley: Explanation of the API terminology, more explicit statement concerning applications that bind to a specific address, and some smaller editorial fixes

Changes compared to version draft-scharf-mptcp-api-02:

- o Definition of the behavior of getpeername() and getsockname() when being called by an MPTCP-aware application.
- o Discussion of the possibility that an MPTCP implementation could support the SCTP API, as far as it is applicable to MPTCP.
- o Various editorial fixes.

Changes compared to version draft-scharf-mptcp-api-01:

- o Second half of the document completely restructured
- o Separation between a basic API and an advanced API: The focus of the document is the basic API only; all text concerning a potential extended API is moved to the appendix

- o Several clarifications, e. g., concerning buffer sizeing and the use of different scheduling strategies triggered by TCP\_NODELAY
- o Additional references

Changes compared to version draft-scharf-mptcp-api-00:

- o Distinction between legacy and MPTCP-aware applications
- o Guidance concerning default enabling, reaction to the shutdown of the first subflow, etc.
- o Reference to a potential use of AF\_MULTIPATH
- o Additional references to related work

#### Authors' Addresses

Michael Scharf  
Alcatel-Lucent Bell Labs  
Lorenzstrasse 10  
70435 Stuttgart  
Germany

EMail: michael.scharf@alcatel-lucent.com

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

EMail: alanford@cisco.com



Internet Engineering Task Force  
Internet-Draft  
Intended status: Experimental  
Expires: December 8, 2012

A. Ford  
Cisco  
C. Raiciu  
University Politehnica of  
Bucharest  
M. Handley  
University College London  
O. Bonaventure  
Universite catholique de  
Louvain  
June 6, 2012

TCP Extensions for Multipath Operation with Multiple Addresses  
draft-ietf-mptcp-multiaddressed-09

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.

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 December 8, 2012.



## Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of 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 . . . . .	4
1.1. Design Assumptions . . . . .	4
1.2. Multipath TCP in the Networking Stack . . . . .	5
1.3. Terminology . . . . .	6
1.4. MPTCP Concept . . . . .	6
1.5. Requirements Language . . . . .	8
2. Operation Overview . . . . .	8
2.1. Initiating an MPTCP connection . . . . .	8
2.2. Associating a new subflow with an existing MPTCP connection . . . . .	9
2.3. Informing the other Host about another potential address . . . . .	9
2.4. Data transfer using MPTCP . . . . .	10
2.5. Requesting a change in a path's priority . . . . .	11
2.6. Closing an MPTCP connection . . . . .	11
2.7. Notable features . . . . .	11
3. MPTCP Protocol . . . . .	12
3.1. Connection Initiation . . . . .	13
3.2. Starting a New Subflow . . . . .	17
3.3. General MPTCP Operation . . . . .	21
3.3.1. Data Sequence Mapping . . . . .	23
3.3.2. Data Acknowledgements . . . . .	26
3.3.3. Closing a Connection . . . . .	27
3.3.4. Receiver Considerations . . . . .	28
3.3.5. Sender Considerations . . . . .	29
3.3.6. Reliability and Retransmissions . . . . .	30
3.3.7. Congestion Control Considerations . . . . .	31
3.3.8. Subflow Policy . . . . .	32
3.4. Address Knowledge Exchange (Path Management) . . . . .	33
3.4.1. Address Advertisement . . . . .	34

3.4.2. Remove Address . . . . .	37
3.5. Fast Close . . . . .	38
3.6. Fallback . . . . .	39
3.7. Error Handling . . . . .	42
3.8. Heuristics . . . . .	43
3.8.1. Port Usage . . . . .	43
3.8.2. Delayed Subflow Start . . . . .	43
3.8.3. Failure Handling . . . . .	44
4. Semantic Issues . . . . .	45
5. Security Considerations . . . . .	46
6. Interactions with Middleboxes . . . . .	47
7. Acknowledgements . . . . .	51
8. IANA Considerations . . . . .	51
9. References . . . . .	53
9.1. Normative References . . . . .	53
9.2. Informative References . . . . .	53
Appendix A. Notes on use of TCP Options . . . . .	54
Appendix B. Control Blocks . . . . .	56
B.1. MPTCP Control Block . . . . .	56
B.1.1. Authentication and Metadata . . . . .	56
B.1.2. Sending Side . . . . .	57
B.1.3. Receiving Side . . . . .	57
B.2. TCP Control Blocks . . . . .	57
B.2.1. Sending Side . . . . .	57
B.2.2. Receiving Side . . . . .	58
Appendix C. Finite State Machine . . . . .	58
Appendix D. Changelog . . . . .	59
D.1. Changes since draft-ietf-mptcp-multiaddressed-05 . . . . .	59
D.2. Changes since draft-ietf-mptcp-multiaddressed-04 . . . . .	59
D.3. Changes since draft-ietf-mptcp-multiaddressed-03 . . . . .	60
D.4. Changes since draft-ietf-mptcp-multiaddressed-02 . . . . .	60
D.5. Changes since draft-ietf-mptcp-multiaddressed-01 . . . . .	60
D.6. Changes since draft-ietf-mptcp-multiaddressed-00 . . . . .	60
D.7. Changes since draft-ford-mptcp-multiaddressed-03 . . . . .	60
D.8. Changes since draft-ford-mptcp-multiaddressed-02 . . . . .	61
Authors' Addresses . . . . .	61

## 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 authors 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.

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 standard 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.

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].

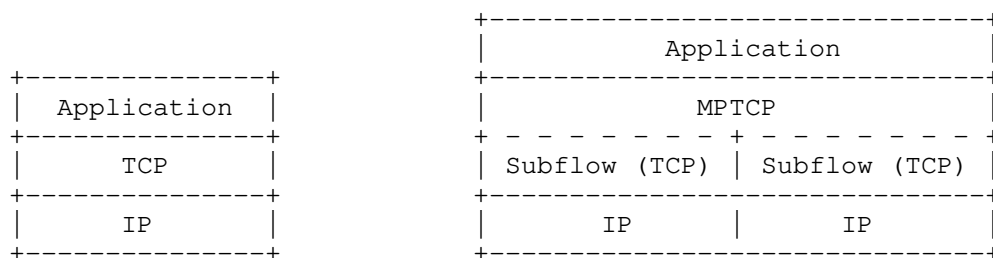


Figure 1: Comparison of Standard TCP and MPTCP Protocol Stacks

### 1.3. Terminology

This document introduces a number of MPTCP-specific terms, defined below:

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

**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.

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.

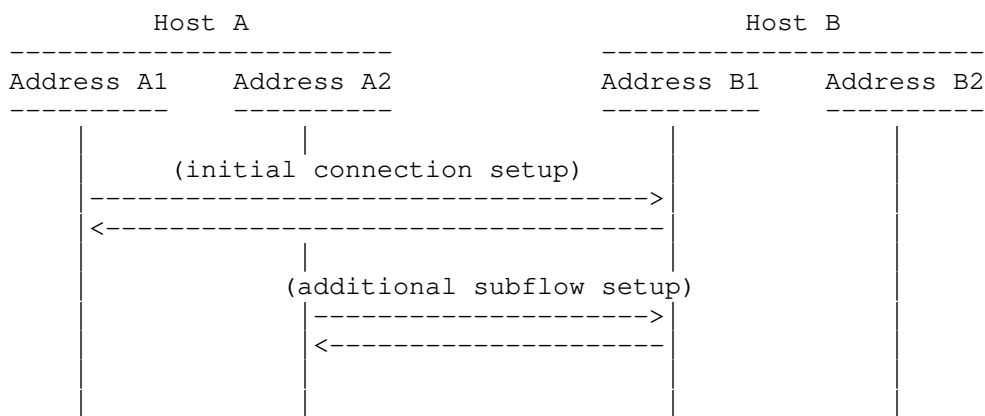


Figure 2: Example MPTCP Usage Scenario

### 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 standard 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. 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 MAC is used for authentication. The MAC 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 MAC, B's nonce,
                                     B's Address ID, flags]

ACK + MP_JOIN                         ->
[A's MAC]
                                     <-
                                     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 acknowledgements 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

acknowledgements for the data sequence space. These acknowledgements 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 [7], the following steps are taken: keys are sent in the clear in the MP\_CAPABLE messages; MP\_JOIN messages are secured with HMAC-SHA1 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.

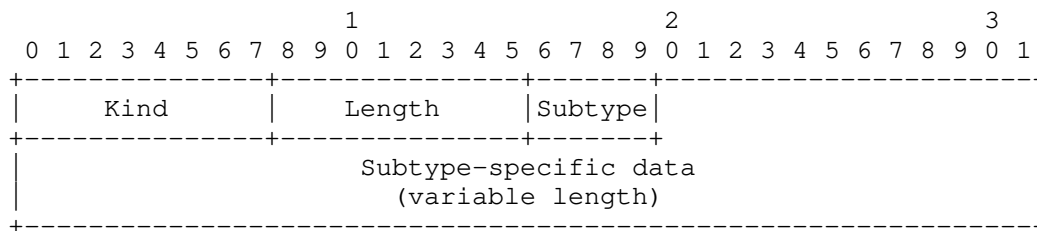


Figure 3: MPTCP option format

Those MPTCP options associated with subflow initiation must be included 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 [8]) 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 [9] 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 signaling purposes, so as 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 [10].

### 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 sender's 64 bit key, which is uniquely linked to 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 token will be a truncated (most significant 32 bits) SHA-1 hash [4]. A different, 64 bit truncation (the least significant 64 bits) of the hash of the key will be used as the Initial Data Sequence Number.

This key is generated by its sender and has local meaning only, 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 keys are given in [11]. Connections will be indexed at each host by the token (the truncated SHA-1 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 very small risk that two different keys will hash to the same token. 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, by checking 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 be closed, 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.
- o SYN/ACK (B->A): B's Key.
- 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 verifiable fashion, allowing 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 SHOULD 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).

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. The leftmost bit - labelled C - 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 may adjust the payload). The remaining bits are used for crypto algorithm negotiation. Currently only the rightmost bit - labeled S - is assigned, and indicates the use of HMAC-SHA1 (as defined in Section 3.2). An implementation that only supports this method MUST set this bit to 1 and all other currently reserved bits to zero. If none of these flags are set, the MP\_CAPABLE option MUST be treated as invalid and ignored (i.e. it must be treated as a regular TCP handshake).

These bits negotiate capabilities in similar ways. For the 'C' bit, 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 C=0. This decision is confirmed by the setting of the 'C' bit in the third packet (the ACK) of the handshake. For example, if the initiator sets C=0 in the SYN, but the responder sets C=1 in the SYN/ACK, checksums MUST be used in both directions, and the initiator will set C=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 S 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.

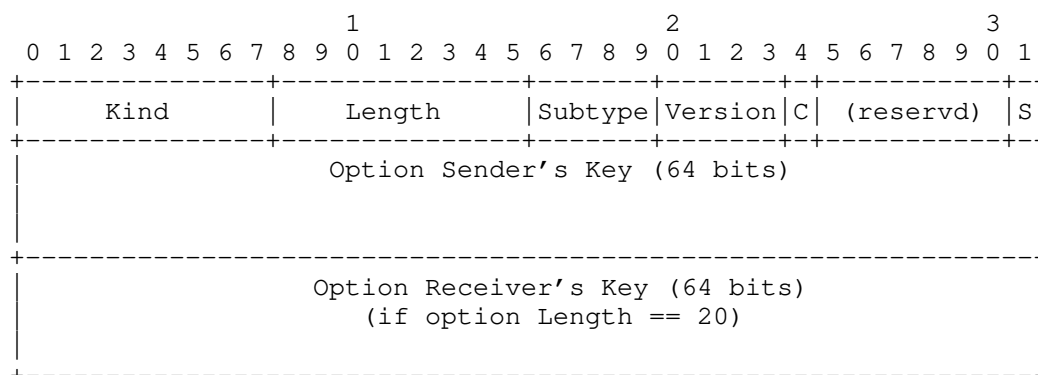


Figure 4: Multipath Capable (MP\_CAPABLE) option

If a SYN contains an MP\_CAPABLE option but the SYN/ACK does not, it is assumed that the passive opener is not multipath capable and thus the MPTCP session 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) is generated as a hash from the Key, i.e.  $IDSN-A = Hash(Key-A)$  and  $IDSN-B = Hash(Key-B)$ . The Hash mechanism here provides the least significant 64 bits of the SHA-1 hash of the key. 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). The tokens presented in this option are generated by the SHA-1 [4] 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).

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.

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.

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.

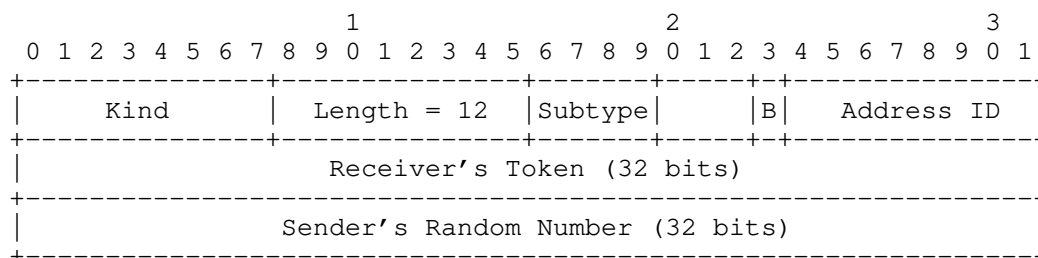


Figure 5: Join Connection (MP\_JOIN) option (for initial SYN)

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) Message Authentication Code (MAC). 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 a MAC 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 MAC 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 MAC exchange at this stage allow both hosts to have first exchanged random data (in the first two SYN packets) that is used as the "message". Both MACs are generated according to HMAC as defined in [12], using the SHA-1 hash algorithm [4] (thus generating a 160-bit / 20 octet HMAC). Due to option space limitations, the MAC included in the SYN/ACK is truncated to the leftmost 64 bits, but this is acceptable since an attacker only has once chance to guess the MAC correctly.

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 MAC 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 MAC 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 MAC 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.

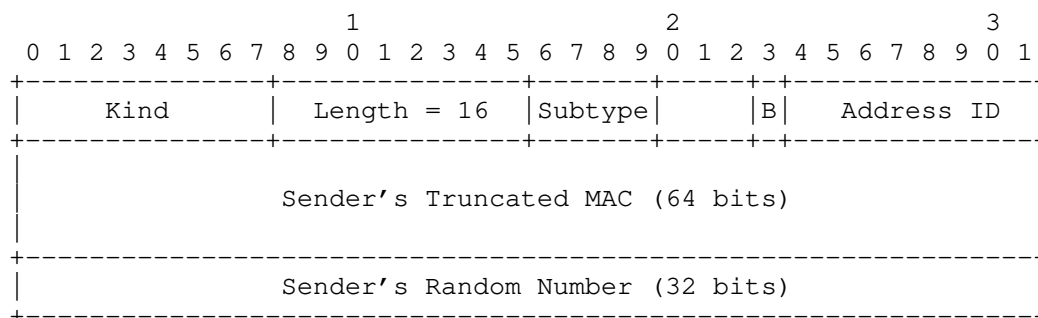


Figure 6: Join Connection (MP\_JOIN) option (for responding SYN/ACK)

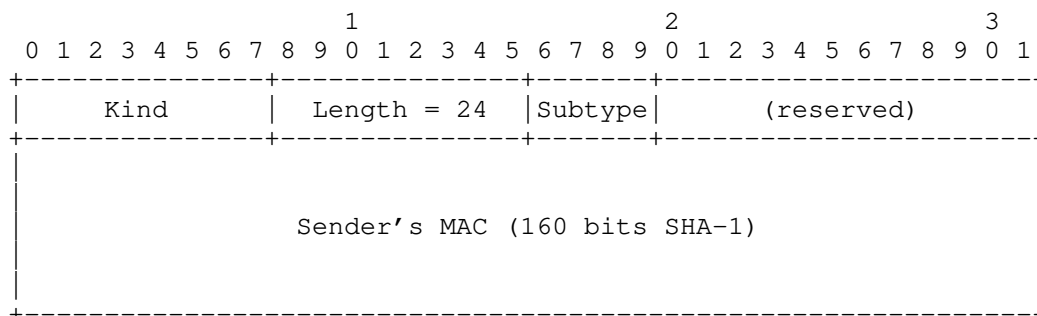
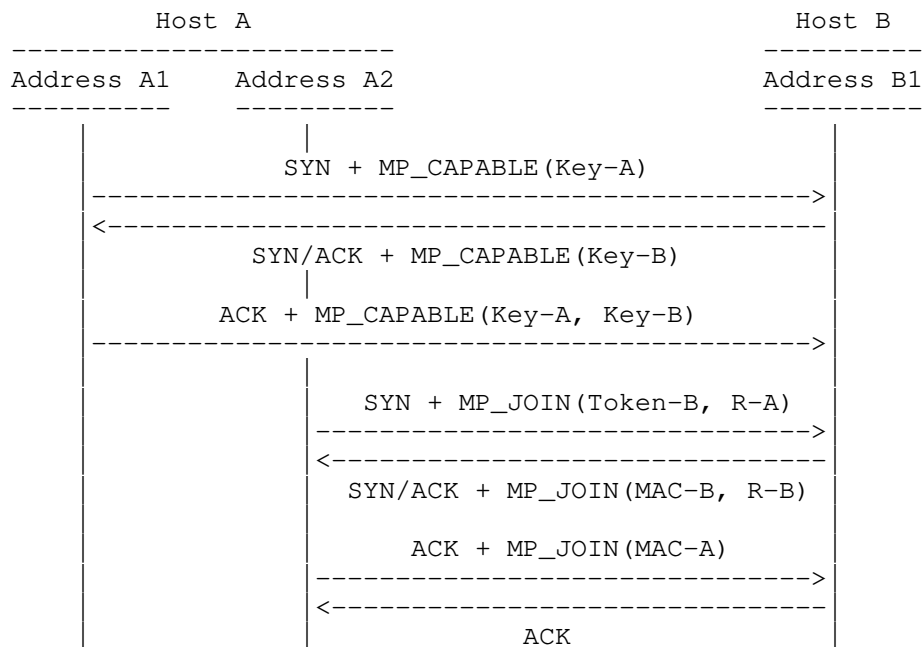


Figure 7: Join Connection (MP\_JOIN) option (for third ACK)

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



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

MAC-B = MAC(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 MAC 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 MAC, or the MP\_JOIN option is missing from the ACK, it MUST close the subflow with a TCP RST.

If the MACs 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.

Deultiplexing 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.

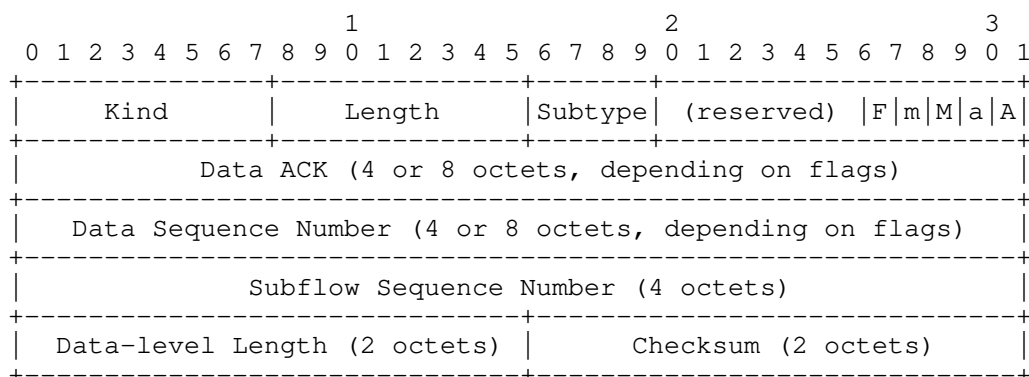


Figure 9: Data Sequence Signal (DSS) option

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. 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, it SHOULD be ignored. If a checksum is not present when its use has been negotiated, the receiver SHOULD 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 [8] 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.

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.

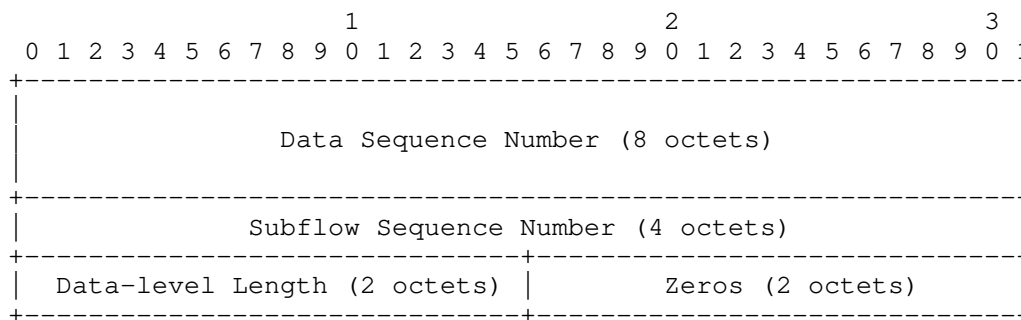


Figure 10: Pseudo-Header for DSS Checksum

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) [13]. 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 is done by 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 Acknowledgements

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 acknowledgements 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 sequence 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:  $SSN - SUBFLOW\_ACK \leq DSN - DATA\_ACK$ , 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. 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<sub>i</sub>*, *ack<sub>i</sub>* + *rcv\_wnd<sub>i</sub>*), where *ack<sub>i</sub>* 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 proactively ACK data.

The sender MUST keep data in its send buffer as long as the data has not been acknowledged at both connection level and on all subflows 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 [14], 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.

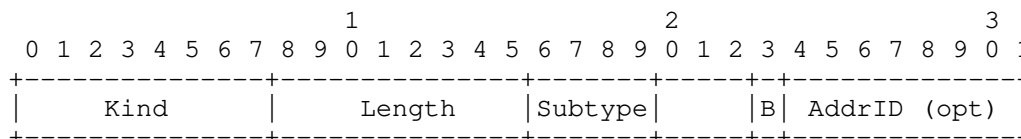


Figure 11: MP\_PRIO option

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 successfully 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.

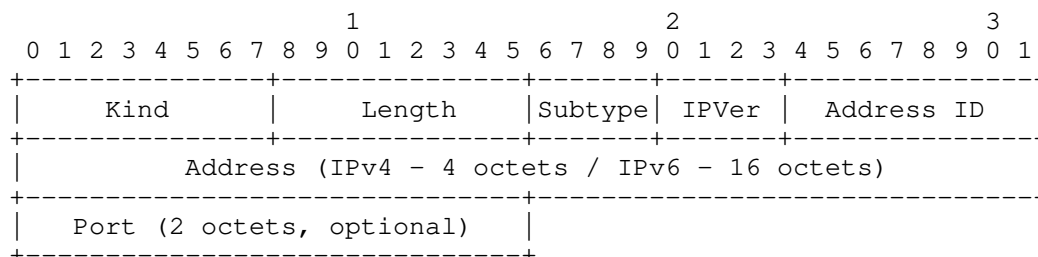


Figure 12: Add Address (ADD\_ADDR) option

Due to the proliferation of NATs, it is reasonably likely that one host may attempt to advertise private addresses [15]. 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. Such advertisements must not, however, cause harm or security vulnerabilities. The standard mechanism to create a new subflow (Section 3.2) 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 MAC 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.

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.

When receiving an ADD\_ADDR message with an Address ID already in use for a live subflow within the connection, 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 [9], 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 [16] 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.

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.

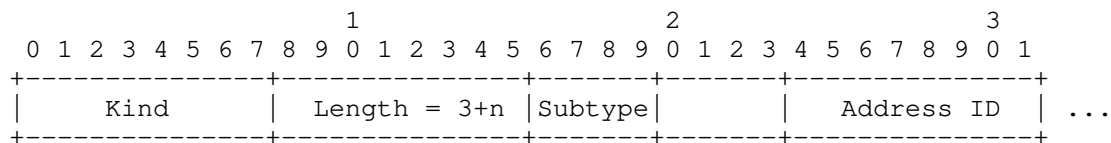


Figure 13: Remove Address (REMOVE\_ADDR) option

### 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.

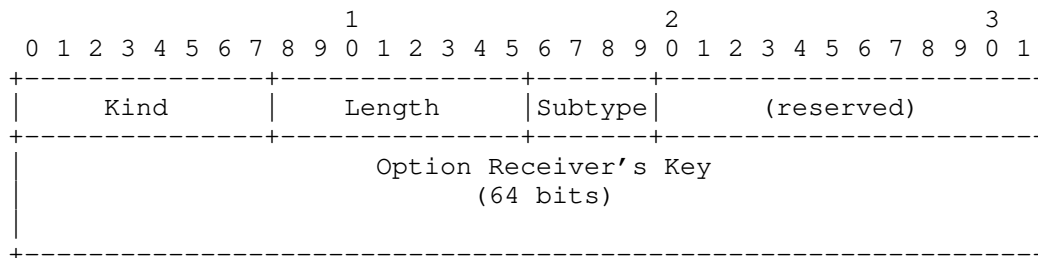


Figure 14: Fast Close (MP\_FASTCLOSE) option

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

- o Host A sends an ACK containing the MP\_FASTCLOSE option on one subflow, containing the key of Host B as declared in the initial

connection handshake. On all the other subflows, Host A sends a regular TCP RST to close these subflows, and tears them down. Host A now enters FASTCLOSE\_WAIT state.

- o Upon receipt of an MP\_FASTCLOSE, containing the valid key, host B answers on the same subflow with a TCP RST and tears down all subflows. Host B can now close the whole MPTCP connection (it transitions directly to CLOSED state).
- o As soon as Host A has received the TCP RST on the remaining subflow, it can close this subflow and tear down the whole connection (transition from FASTCLOSE\_WAIT to CLOSED states). If Host A receives an MP\_FASTCLOSE instead of a TCP RST, both hosts attempted fast closure simultaneously. Host A should reply with a TCP RST and tear down the connection.
- o If host A does not receive a TCP RST in reply to its MP\_FASTCLOSE after one 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.

### 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.

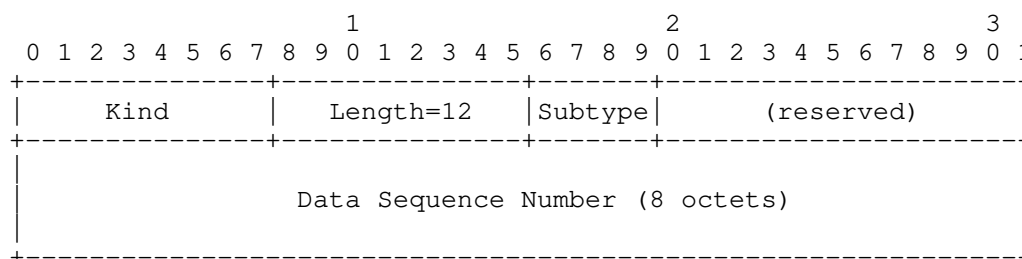


Figure 15: Fallback (MP\_FAIL) option

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, referring to the data-level sequence number of the start of the segment on which the checksum error was detected. The sender will receive this, and if all unacknowledged data in flight is contiguous, will signal an infinite mapping. This infinite mapping will be a DSS option (Section 3.3) on the first new packet, containing a Data Sequence Mapping that acts retroactively, referring to the start of the subflow sequence number of the last segment that was known to be delivered intact. From that point onwards data can be altered by a middlebox without affecting MPTCP, as the data stream is equivalent to a regular, legacy TCP session.

In the rare case that the data is not contiguous (which could happen when there is only one subflow but it is retransmitting data from a subflow that has recently been uncleanly closed), the receiver MUST close the subflow with 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 receiver 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 MAC 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).

#### 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 [7], 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. To ease demultiplexing whilst not giving away any cryptographic material, future subflows use a truncated SHA-1 hash of this key as the connection identification "token". The keys are concatenated and used as keys for creating Message Authentication Codes (MAC) 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 MAC will never be the same on two handshakes.

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 [7].

## 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.

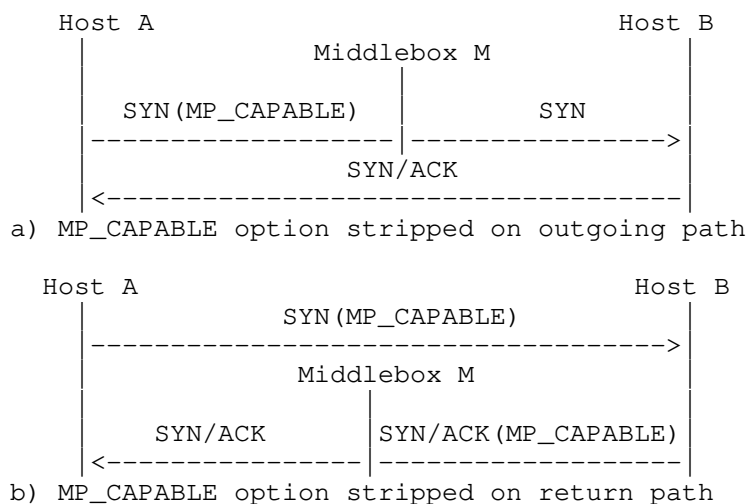


Figure 16: Connection Setup with Middleboxes that Strip Options from

## 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 [17] (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 [15] 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) [18] 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 [19] 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 [20] 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. Acknowledgements

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. 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.

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, and Anumita Biswas.

## 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 MPTCP option. Initial values for the MPTCP option subtype registry are given below; future assignments are to be defined by RFCs (RFC Requirement as defined by [21]) 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 ACK and Data Sequence Mapping)	Section 3.3	0x2
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.

This document also requests that IANA keeps a registry of "MPTCP cryptographic handshake algorithms" based on the flags in MP\_CAPABLE (Section 3.1). This document specifies only one algorithm:

Flags	Algorithm Name	Reference
0x1	HMAC-SHA1	This document, Section 3.2

Table 3: MPTCP Handshake Algorithms

Future assignments in this registry are also to be defined by RFCs (RFC Requirement as defined by [21]) Assignments consist of the value of the flags, a symbolic name for the algorithm, and a reference to its specification.

Note that the length of this field is not fixed; it is a definition of the meaning of each bit in this field (i.e. 0x2, 0x4, 0x8, 0x10, etc). Future specifications may define additional flags using the leftmost bits of this field, and therefore the number of bits available for cryptographic negotiation may change.

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

## 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] Bagnulo, M., "Threat Analysis for TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6181, March 2011.
- [8] Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, "TCP Selective Acknowledgment Options", RFC 2018, October 1996.
- [9] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, September 2009.
- [10] Gont, F., "Survey of Security Hardening Methods for Transmission Control Protocol (TCP) Implementations", draft-ietf-tcpm-tcp-security-03 (work in progress), March 2012.
- [11] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.
- [12] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.
- [13] Jacobson, V., Braden, B., and D. Borman, "TCP Extensions for High Performance", RFC 1323, May 1992.
- [14] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of

Explicit Congestion Notification (ECN) to IP", RFC 3168, September 2001.

- [15] Rekhter, Y., Moskowitz, R., Karrenberg, D., Groot, G., and E. Lear, "Address Allocation for Private Internets", BCP 5, RFC 1918, February 1996.
- [16] Braden, R., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, October 1989.
- [17] Srisuresh, P. and K. Egevang, "Traditional IP Network Address Translator (Traditional NAT)", RFC 3022, January 2001.
- [18] Border, J., Kojo, M., Griner, J., Montenegro, G., and Z. Shelby, "Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations", RFC 3135, June 2001.
- [19] 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](http://www.usenix.org/events/sec01/full_papers/handley/handley.pdf)>.
- [20] Freed, N., "Behavior of and Requirements for Internet Firewalls", RFC 2979, October 2000.
- [21] 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.

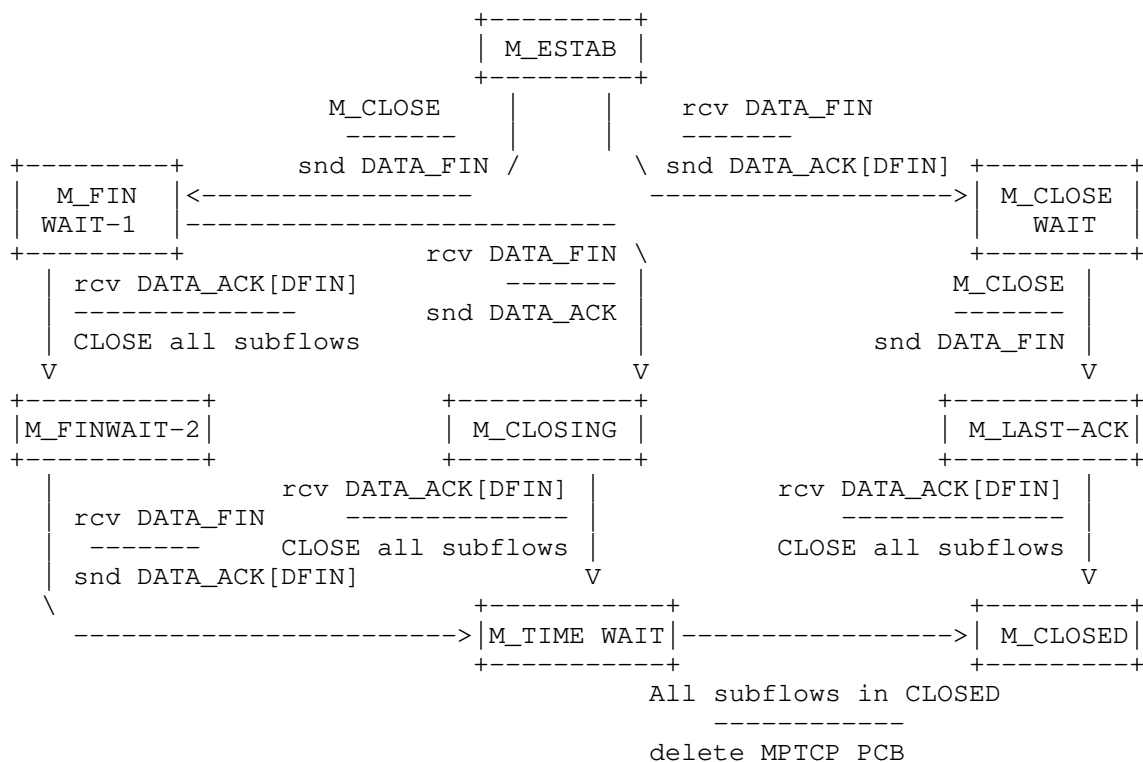


Figure 17: Finite State Machine for Connection Closure

## Appendix D. Changelog

(To be removed by the RFC Editor)

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

## D.1. Changes since draft-ietf-mptcp-multiaddressed-05

- o Added MP\_FASTCLOSE mechanism.

## D.2. Changes since draft-ietf-mptcp-multiaddressed-04

- o Reverted change to MP\_CAPABLE from last revision.
- o Clarifications in response to comments.

## D.3. Changes since draft-ietf-mptcp-multiaddressed-03

- o Removed Key from MP\_CAPABLE on SYN (it is in the ACK).
- o Added optional Address ID to MP\_PRIO.
- o Responded to review comments.

## D.4. Changes since draft-ietf-mptcp-multiaddressed-02

- o Changed to using a single TCP option with a sub-type field.
- o Merged Data Sequence Number, DATA\_ACK, and DATA\_FIN.
- o Changed DATA\_FIN behaviour (separated from subflow FIN).
- o Added crypto agility and checksum negotiation.
- o Redefined MP\_JOIN handshake to use only three TCP options.
- o Added pseudo-header to checksum.
- o Many clarifications and re-structuring.
- o Added more discussion on heuristics.

## D.5. Changes since draft-ietf-mptcp-multiaddressed-01

- o Added proposal for hash-based security mechanism.
- o Added receiver subflow policy control (backup path flags and MP\_PRIO option).
- o Changed DSN\_MAP checksum to use the TCP checksum algorithm.

## D.6. Changes since draft-ietf-mptcp-multiaddressed-00

- o Various clarifications and minor re-structuring in response to comments.

## D.7. Changes since draft-ford-mptcp-multiaddressed-03

- o Clarified handshake mechanism, especially with regard to error cases (Section 3.2).
- o Added optional port to ADD\_ADDR and clarified situation with private addresses (Section 3.4.1).

- o Added path liveness check to REMOVE\_ADDR (Section 3.4.2).
- o Added chunk checksumming to DSN\_MAP (Section 3.3.1) to detect payload-altering middleboxes, and defined fallback mechanism (Section 3.6).
- o Major clarifications to receive window discussion (Section 3.3.5).
- o Various textual clarifications, especially in examples.

#### D.8. Changes since draft-ford-mptcp-multiaddressed-02

- o Remove Version and Address ID in MP\_CAPABLE in Section 3.1, and make ISN be 6 bytes.
- o Data sequence numbers are now always 8 bytes. But in some cases where it is unambiguous it is permissible to only send the lower 4 bytes if space is at a premium.
- o Clarified behaviour of MP\_JOIN in Section 3.2.
- o Added DATA\_ACK to Section 3.3.
- o Clarified fallback to non-multipath once a non-MP-capable SYN is sent.

#### 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](mailto: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](mailto:olivier.bonaventure@uclouvain.be)



Multipath TCP  
INTERNET-DRAFT  
Intended Status: Standard Track  
Expires: January 10, 2013

Ramin Khalili  
Nicolas Gast  
Miroslav Popovic  
Jean-Yves Le Boudec  
EPFL-LCA2  
July 9, 2012

<Performance Issues with MPTCP>  
draft-khalili-mptcp-performance-issues-00

Abstract

We show, by measurements over a testbed and by mathematical analysis, that the current MPTCP suffers from two problems: (P1) Upgrading some TCP users to MPTCP can reduce the throughput of others without any benefit to the upgraded users; and (P2) MPTCP users can be excessively aggressive towards TCP users. We attribute these problems to the "Linked Increases" Algorithm (LIA) of MPTCP [4], and more specifically, to an excessive amount of traffic transmitted over congested paths. Our results show that these problems are important and can be mitigated. We believe that the design of the congestion control of MPTCP should be improved.

Status of this Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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

This Internet-Draft will expire on January 10, 2013.

The list of current Internet-Drafts can be accessed at  
<http://www.ietf.org/lid-abstracts.html>

The list of Internet-Draft Shadow Directories can be accessed at  
<http://www.ietf.org/shadow.html>

## Copyright and License Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of 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 . . . . .	3
1.1	Requirements Language . . . . .	4
1.2	Terminology . . . . .	4
2	MPTCP's LIA . . . . .	5
3	Testbed Setup . . . . .	6
4	Scenario A: MPTCP can penalize regular TCP users . . . . .	7
5	Scenario B: MPTCP can penalize other MPTCP users . . . . .	10
6	Scenario C: MPTCP is excessively aggressive towards TCP users . . . . .	12
7	Can the suboptimality of MPTCP with LIA be fixed? . . . . .	14
8	Conclusion . . . . .	16
9	References . . . . .	16
9.1	Normative References . . . . .	16
9.2	Informative References . . . . .	16
	Authors' Addresses . . . . .	18



## 1 Introduction

Regular TCP uses a window-based congestion-control mechanism to adjust the transmission rate of users [2]. It always provides a Pareto-optimal allocation of resources: it is impossible to increase the throughput of one user without decreasing the throughput of another or without increasing the congestion cost [5]. It also guarantees a fair allocation of bandwidth among the users but favors the connections with lower RTTs [6].

Various mechanisms have been used to build a multipath transport protocol compatible with the regular TCP. Inspired by utility maximization frameworks, [7, 8] propose a family of algorithms. These algorithms tend to use only the best paths available to users and are optimal in static settings where paths have similar RTTs. In practice, however, they suffer from several problems [9]. First, they might fail to quickly detect free capacity as they do not probe paths with high loss probabilities sufficiently. Second, they exhibit flappiness: When there are multiple good paths available to a user, the user will randomly flip its traffic between these paths. This is not desirable, specifically, when the achieved rate depends on RTTs, as with regular TCP.

Because of the issues just mentioned, the congestion control part of MPTCP does not follow the algorithms in [7, 8]. Instead, it follows an ad-hoc design based on the following goals [4]. (1) Improve throughput: a multipath TCP user should perform at least as well as a TCP user that uses the best path available to it. (2) Do no harm: a multipath TCP user should never take up more capacity from any of its paths than a regular TCP user. And (3) balance congestion: a multipath TCP algorithm should balance congestion in the network, subject to meeting the first two goals.

MPTCP compensates for different RTTs and solves many problems of multipath transport [10, 11]: It can effectively use the available bandwidth, it improves throughput and fairness compared to independent regular TCP flows in many scenarios, and it solves the flappiness problem.

We show, however, by measurements over our testbed and mathematical analysis, that MPTCP still suffers from the following problems:

(P1) Upgrading some regular TCP users to MPTCP can reduce the throughput of other users without any benefit to the upgraded users. This is a symptom of non-Pareto optimality.

(P2) MPTCP users can be excessively aggressive towards TCP users.

We attribute these problems to the "Linked Increases" Algorithm (LIA) of MPTCP [4] and specially to an excessive amount of traffic transmitted over congested paths.

These problems indicate that LIA fails to fully satisfy its design goals, especially goal number 3. The design of LIA forces a trade off between optimal resource pooling and responsiveness, it cannot provide both at the same time. Hence, to provide good responsiveness, LIA's current implementation must depart from Pareto-optimality, which leads to problems (P1) and (P2).

This document provides a number of examples and scenarios (Sections 4 to 6) in which MPTCP with LIA exhibits problems (P1) and (P2). Our results show that the identified problems with LIA are important. Moreover, we show in Section 7 that these problems are not due to the nature of a window-based multipath protocol, but rather to the design of LIA; it is possible to build an alternative to LIA that mitigates these problems and is as responsive and non-flappy as LIA. Hence, we believe that the design of the congestion control of MPTCP should be improved.

## 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 [1].

## 1.2 Terminology

Regular TCP: The standard version of TCP that operates between a single pair of IP addresses and ports [2].

Multipath TCP: A modified version of the regular TCP that allows a user to spread its traffic across multiple paths.

MPTCP: The proposal for multipath TCP specified in [3].

LIA: The "Linked Increases" Algorithm of MPTCP (the congestion control of MPTCP) [4].

OLIA: The Opportunistic "Linked Increases" Algorithm for MPTCP proposed in our technical report [12].

RTT: The Round-Trip Time seen by a user.

MSS: The Maximum Segment Size that specifies the largest amount of data can be transmitted by a TCP packet.

AP: Access Point.

## 2 MPTCP's LIA

MPTCP's congestion-control algorithm forces a trade-off between optimal resource pooling and responsiveness [9]. The idea behind the algorithm is to transmit over a path  $r$  at a rate proportional to  $p_r^{-1/\epsilon}$ , where  $p_r$  is the loss probability over this link and  $\epsilon$  is a design parameter.

The value of  $\epsilon$  is between 0 and 2. The choice  $\epsilon=0$  corresponds to the fully coupled algorithm of [7, 8]: the traffic is sent only over the best paths, it is Pareto-optimal but flappy. The choice  $\epsilon=2$  corresponds to having uncoupled TCP flow on each path: it is very responsive and non-flappy but does not balance congestion.

MPTCP's implementation uses  $\epsilon=1$  to provide a compromise between optimal resource pooling and responsiveness. This algorithm is called "Linked Increases" Algorithm (LIA) [4].

### 3 Testbed Setup

To investigate the behavior of MPTCP in practice, three testbed topologies are created representing scenarios in Sections 4, 5, and 6. We use server-client PCs that run MPTCP enabled Linux kernels. We use MPTCP for the Linux kernel 3.0.0 released in February 2012. In all our scenarios, laptop PCs are used as routers. We install "Click Modular Router" software [13] on the routers to implement topologies with different characteristics. Iperf is used to create multiple connections.

In our scenarios, we are able to implement links with configurable bandwidth and delay. We are also able to set the parameters of the RED queues following the structure in [14]. For a 10 Mbps link, we set the packet loss probability equal to 0 up to a queue size of 25 Maximum Segment Size (MSS). Then it grows linearly to the value 0.1 at 50 MSS. It again increases linearly up to 1 for 100 MSS. The parameters are proportionally adapted when the link capacity changes.

To verify that the problems observed are caused by the congestion-control algorithm of MPTCP and not by some unknown problems in our testbed, we perform a mathematical analysis of MPTCP. This analysis is based on the fix point analysis of MPTCP. As we will see, our mathematical results confirm our measurement results. The details of these mathematical analyses are available in [12].

#### 4 Scenario A: MPTCP can penalize regular TCP users

Consider a network with two types of users. There are  $N_1$  users of type1, each with a high-speed private connection. These users access different files on a media streaming server. The server has an access rate limit of  $N_1C_1$  Mbps. Type1 users can activate a second connection through a shared AP by using MPTCP. There are also  $N_2$  users of type2; they are connected to the shared AP. They download their contents from the Internet. The shared AP has a capacity of  $N_2C_2$  Mbps.

We implement this scenario in a testbed similar to Figure 1. Within router-PCs R1 and R2, we implement links with capacities  $N_1C_1$  and  $N_2C_2$  and RTTs of 150 ms (including queuing delay), modeling the bottleneck at the server side and the shared AP, respectively. High speed connections are used to implement private connections of type1 users.

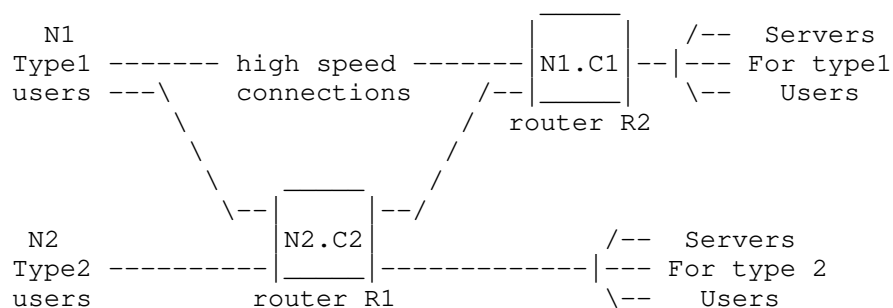


Figure 1: Testbed implementation of Scenario A: router R1 implement the streaming server and router R2 the shared AP.

When type1 users use only their private connection, each type1 user receives a rate of  $C_1$  and each type2 user receives a rate of  $C_2$ . By upgrading type1 users to MPTCP, we observe, through measurement and by mathematical analysis, that the throughput of type2 users significantly decreases. However, type1 users receive the same throughput as before, because the bottleneck for their connections is at the server side. We report the throughput received by users after upgrading type1 users to MPTCP on Table 1 for  $C_1=C_2=1$  Mbps. For each case, we take 5 measurements. In each case, the confidence intervals are very small (less than 0.01Mbps).

In [12 Section3.2], we provide a mathematical analysis of MPTCP that confirm our measurements: The predicted rate of type1 users is always 1 and the predicted rate for type2 users is, respectively, 0.74 when  $N_1=N_2=10$  and 0.50 when  $N_1=30$  and  $N_2=10$ .

		Type1 users are single path (measurement)	Type1 users are multipath		
			MPTCP (meas.)	optimal algorithm with p. cost (theory)	w/out p. cost (theory)
N1=10	type1	0.98	0.96	1	1
N2=10	type2	0.98	0.70	0.94	1
N1=30	type1	0.98	0.98	1	1
N2=10	type2	0.98	0.44	0.8	1

meas.=measurements, p.=probing, w/out=without, Values are in Mbps.

Table 1: Throughput obtained by type1 and type2 users in Scenario A: upgrading type1 users to MPTCP decrease the throughput of type2 with not benefit for type1 users. The problem is much less critical using optimal algorithm with probing cost.

We observe that MPTCP exhibits problem (P1): upgrading type1 users to MPTCP penalizes type2 users without any gain for type1 users. As the number of type1 users increases, the throughput of type2 users decreases, but the throughput of type1 users does not change as it is limited by the capacity of the streaming server. For  $N1=N2$ , type2 users see a decrease of 30%. When  $N1=3N2$ , this decrease is 55%.

We compare the performance of MPTCP with two theoretical baselines. They serve as references to see how far from the optimum MPTCP with LIA is. We show in Section 7 that it is possible to replace LIA by an alternative that keeps the same non-flappiness and responsiveness and performs closer to the optimum.

The first baseline is an algorithm that provides theoretical optimal resource pooling in the network (as discussed in [7] and several other theoretical papers). We refer to it as "optimal algorithm without probing cost".

In practice, however, the value of the congestion windows are bounded below by 1 MSS. Hence, with a window-based congestion-control algorithm, a minimum probing traffic of 1 MSS per RTT will be sent over a path. We introduce a second theoretical baseline, called "optimal algorithm with probing cost"; it provides optimal resource pooling in the network given that a minimum probing traffic of 1 MSS per RTT is sent over each path.

We show the performance of these optimal algorithms in Table 1. Using an optimal algorithm with probing cost, the entire capacity of the shared AP is allocated to type2 users. Hence, all the users in the network receives a throughput of 1 Mbps. By using an optimal algorithm with probing cost, type1 users will send only 1MSS per RTT over the shared AP. Hence, we observe a decrease on the throughput of type2 users. However, the decrease is much less than what we observe using MPTCP. The performance of our proposed alternative to LIA is shown in Section 7, Table 4.

This performance problem of MPTCP can be explained by the fact that LIA does not fully balance congestion. For  $N1=N2$ , we observe through measurements that  $p1=0.009$  and  $p2=0.02$  (the probability of losses at routers R1 and R2). For  $N1=3N2$ , the value of  $p1$  remains almost the same and  $p2$  increases to  $p2=0.05$ . LIA excessively increases congestion on the shared AP, which is not in compliance with goal 3. In [12], we propose an alternative to LIA. Using this algorithm, we have  $p1=0.009$  and  $p2=0.012$  for  $N1=10$  and  $0.018$  for  $N1=30$ . Hence, it is possible to provide a better congestion balancing in the network.

## 5 Scenario B: MPTCP can penalize other MPTCP users

Consider a multi-homing scenario as follows. We have four Internet Service Providers (ISPs), X, Y, Z, and T. Y is a local ISP in a small city, which connects to the Internet through Z. X, Z, and T are nation-wide service providers and are connected to each other through high speed links. X provides Internet services to users in the city and is a competitor of Y. They have access capacity limits of CX, CY, CZ, and CT. Z and T are hosts of different video streaming servers.

There are two types of users: Blue users download contents from servers in Z and Red users download from servers in ISP T. To increase their reliability, Blue users use multi-homing and are connected to both ISPs X and Y. Red users can connect either only to Y or to both X and Y.

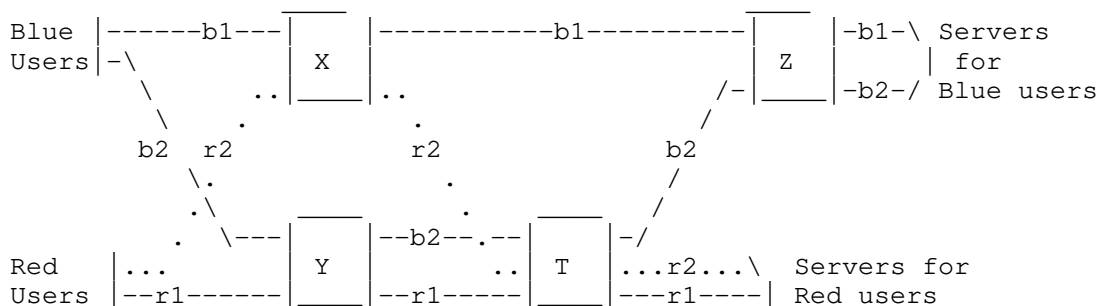


Figure 2. Testbed implementation of Scenario B. Blue users transmit over path b1 and b2. Red users use path r1, but can upgrade to MPTCP by establishing a second connection through path r2.

We implement the scenario in a testbed similar to Figure 2. b1 and b2 are the paths available to Blue users. Red users use the path r1, but can upgrade to MPTCP by establishing a second connection through path r2. The measurement results are reported in Table 2 for a setting with CX=27, CT=36, and CY=CZ=100, all in Mbps, where we have 15 Red and 15 Blue users. RTTs are around 150 ms (including queuing delay) over all paths. We also show the performance of theoretically optimal algorithms with and without probing cost.

We observe that when Red users only connect to ISP Y, the aggregate throughput of users is close to the cut-set bound, 63 Mbps. However, Blue users get a higher share of the network bandwidth. Now, let's consider that Red users upgrade to MPTCP by establishing a second connection through X (showed by pointed-line in Figure 2). Our results in Table 1 show that Red users do not receive any higher throughput. However, the average rate of Blue users drops by 20%,



which results in a drop of 13% in aggregate throughput.

	Red users are single-path			Red users are multipath		
	MPTCP (meas.)	Blue users use optimal algorithm with p. cost (theory)	w/out p. cost (theory)	MPTCP (meas.)	Blue and Red users use optimal algorithm with p. cost (theory)	w/out p. cost (theory)
Red users	1.5	2.1	2.1	1.4	2.04	2.1
Blue users	2.5	2.1	2.1	2.0	2.04	2.1

meas.=measurements, p.=probing, w/out=without, Values are in Mbps.

Table 2: Throughput received by users before and after upgrading Red users to MPTCP. We have 15 Red and 15 Blue users. By upgrading Red users to MPTCP, the aggregate throughput of users decreases by 13% with no benefit for Red users.

In [12 Section 3.3], we also provide a mathematical analysis of MPTCP. Our mathematical results predict that by upgrading the Red user to MPTCP the rate of Blue users will be reduced by 21%. This results in 14% decrease in the aggregate throughput. Hence, our mathematical results confirm our observations from the measurement. Similar behavior is predicted for other values of CX and CT [12 Figure 4a].

Using an optimal algorithm without probing cost, Red users transmit only over path r1 and Blue users split their traffic over paths b1 and b2 to equalize the rate of blue and red users. Upgrading Red users to multipath does not change the allocation. Hence, we observe no decrease in the aggregate throughput and the rate of each user. By using an optimal algorithm with probing cost, the rate of Blue and Red users decreases by 3% when we upgrade the Red users to multipath users since red users are forced to send 1 MSS per RTT over path r2. This decrease is much less than what we observe using MPTCP with LIA. The performance of our proposed alternative to LIA is shown in Section 7, Table 5.

Similarly to Scenario A, the problem can be attributed to the excessive amount of traffic sent over the congested paths. This illustrates that MPTCP fails to balance the congestion in the network.

## 6 Scenario C: MPTCP is excessively aggressive towards TCP users

Consider a scenario with  $N_1$  multipath users,  $N_2$  single-path users, and two APs with capacities  $N_1C_1$  and  $N_2C_2$  Mbps. Multipath users connect to both APs and they share AP2 with single-path users. The users download their contents from the Internet. This scenario is implemented in a testbed similar to Figure 3.

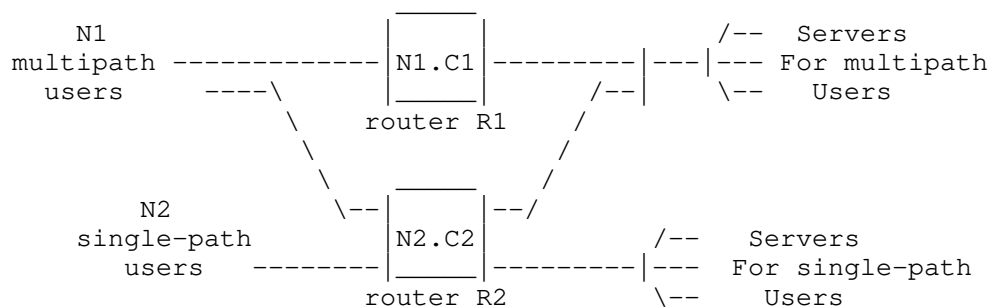


Figure 3: Testbed implementation of Scenario C: routers R1 and R2 implement AP1 and AP2 with capacities  $N_1C_1$  and  $N_2C_2$  Mbps.

If the allocation of rates is proportionally fair, multipath users will use AP2 only if  $C_1 < C_2$  and all users will receive the same throughput. When  $C_1 > C_2$ , a fair multipath user will not transmit over AP2. However, using MPTCP with LIA, multipath users receive a disproportionately larger share of bandwidth.

We implement the scenario in our testbed and measure the performance of MPTCP with LIA. We report the throughput received by single-path and multipath users in Table 3. We present the results for  $N_2=10$  and two values of  $N_1=10$  and 30, where  $C_1=C_2=1$  Mbps. RTTs are around 150 ms (including queuing delay). We also present the performance of optimal (proportionally fair) algorithms.

As  $C_1=C_2$ , for any fairness criterion, multipath users should not use AP2. Our results show that, MPTCP users are disproportionately aggressive and exhibit problem (P2). Hence, single-path users receive a much smaller share than they should. For  $N_1=N_2$ , single-path users see a decrease of about 30% in their received throughput compared to a fair allocation. When  $N_1=3N_2$ , this decrease is around 55%.

These measurements are confirmed by our mathematical analysis as shown in [12 Section 3.4]. The predicted rate of type1 users is 1.3 for  $N_1=N_2=10$  and is 1.17 when  $N_1=30$  and  $N_2=10$ . For type2 users, the predicted rate is 0.7 when  $N_1=N_2=10$  and 0.48 when  $N_1=30$  and  $N_2=10$ .

		multipath users use MPTCP (measurement)	multipath users use optimal algorithm with p. cost (theory)	w/out p. cost (theory)
N1=10	multipath users	1.3	1.04	1
N2=10	single-path users	0.68	0.94	1
N1=30	multipath users	1.19	1.04	1
N2=10	single-path users	0.38	0.8	1

p.=probing, w/out=without, Values are in Mbps.

Table3: Throughput obtained by single-path and multipath users in Scenario C: MPTCP is excessively aggressive toward TCP users and performs far from how an optimal algorithm would perform.

An optimal algorithm with probing cost provide a proportional fairness among the users. By using an optimal algorithm with probing cost, single-path users receive a rate less than what a proportionally fair algorithm will provide them. However, as we observe, the problem is much less critical compared to the case we use MPTCP. The performance of our proposed alternative to LIA is shown in Section 7, Table 6.

These results clearly show that MPTCP suffers from fairness issues. The problem occurs because LIA fails to fully satisfy goal 3. As in Scenarios A and B, MPTCP sends an excessive amount of traffic over the congested paths.

## 7 Can the suboptimality of MPTCP with LIA be fixed?

We have shown in Section 4, 5, and 6 that MPTCP with LIA performs far behind an optimal algorithm. The question is, "is it possible to modify the congestion control algorithm of MPTCP to perform closer to the optimum". To answer this question, we implement a new congestion control algorithm for MPTCP, called Opportunistic "Linked Increases" Algorithm (OLIA). In this section, we show that in Scenarios A, B and C OLIA performs close to an optimal algorithm with probing cost. Moreover, as shown in [12, Sections 4.3 and 6.2], OLIA keeps the same non-flappiness and responsiveness as LIA.

Contrary to LIA, OLIA's design is not based on a trade-off between responsiveness and optimal resource pooling. It is rooted in the optimal algorithm of [7] with a term that makes it as responsive and non-flappy as LIA. We implemented OLIA by modifying the congestion control part of the MPTCP implementation based on the Linux Kernel 3.0.0. For conciseness, we do not describe OLIA in this paper and refer to [12] for details about the algorithm and its implementation.

We study the performance of MPTCP with OLIA through measurements in Scenarios A, B, and C. The results are reported in Tables 4, 5 and 6. We compare the performance of our algorithm with MPTCP with LIA and with optimal algorithms. We observe that in all cases, MPTCP with OLIA provide a significant improvement over MPTCP with LIA. Moreover, it performs close to an optimal algorithm with probing cost.

		Type1 users are single path (measurement)	Type1 users are multipath		
			MPTCP w. OLIA [with LIA] (measurement)	optimal with p. cost (theory)	algorithm w/out p. cost (theory)
N1=10	type1	0.98	0.98 [0.96]	1	1
N2=10	type2	0.98	0.86 [0.70]	0.94	1
N1=30	type1	0.98	0.98 [0.98]	1	1
N2=10	type2	0.98	0.75 [0.44]	0.8	1

p.=probing, w.=with, w/out=without, Values are in Mbps.

Table 4. Performance of MPTCP with OLIA compared to MPTCP with LIA in scenario A. We show the throughput obtained by users before and after upgrading type1 users to MPTCP. The values in brackets are the values for MPTCP with LIA (taken from table 1).

	Red users are single-path			Red users are multipath		
	MPTCP with OLIA [with LIA] (meas.)	Blue users use optimal algorithm with p. cost (theory)	w/out p. cost (theory)	MPTCP with OLIA [with LIA] (meas.)	Blue and Red users use optimal algorithm with p. cost (theory)	w/out p. cost (theory)
Red users	1.8 [1.5]	2.1	2.1	1.7 [1.4]	2.04	2.1
Blue users	2.2 [2.5]	2.1	2.1	2.2 [2.0]	2.04	2.1

meas.=measurements, p.=probing, w/out=without, Values are in Mbps.

Table 5. Performance of MPTCP with OLIA compared to MPTCP with LIA in scenario B. We show the throughput received by users before and after upgrading Red users to MPTCP. The values in brackets are the values for MPTCP with LIA (taken from Table 2).

		multipath users use MPTCP with OLIA [with LIA] (measurement)		multipath users use optimal algorithm with probing cost (theory)		w/out probing cost (theory)	
N1=10	multipath users	1.11	[1.30]	1.04		1	
N2=10	single-path users	0.88	[0.68]	0.94		1	
N1=10	multipath users	1.1	[1.19]	1.04		1	
N2=10	single-path users	0.72	[0.38]	0.8		1	

meas.=measurements, w/out=without, Values are in Mbps.

Table 6. Performance of MPTCP with OLIA compared to MPTCP with LIA in scenario C. We show the throughput obtained by single-path and multipath users. The values in brackets are the values for MPTCP with LIA (taken from Table 3).

The results show that it is possible to perform close to an optimal algorithm with probing cost by using a TCP-like algorithm. Moreover, we show in [12, Section 4.3 and Section 6.2] that MPTCP with OLIA is as responsive and non-flappy as MPTCP with LIA. This shows that it is possible to build a practical multi-path congestion control that works close to an optimal algorithm with probing cost.

## 8 Conclusion

We have shown that MPTCP with LIA suffers from important performance issues. Moreover, it is possible to build an alternative to LIA that performs close to an optimal algorithm with probing cost while being as responsive and non-flappy as LIA. Hence, we believe that IETF should revisit the congestion control part of MPTCP and that an alternative algorithm, such as OLIA [12], should be considered.

## 9 References

### 9.1 Normative References

- [1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [2] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, September 2009.
- [3] Ford, A., Raiciu, C., Greenhalgh, A., and M. Handley, "Architectural Guidelines for Multipath TCP Development", RFC 6182, March 2011.
- [4] Raiciu, C., Handley, M., and D. Wischik, "Coupled Congestion Control for Multipath Transport Protocols", RFC 6356, October 2011.

### 9.2 Informative References

- [5] F.P. Kelly. Mathematical modelling of the internet. Mathematics unlimited-2001 and beyond.
- [6] F.P. Kelly, A.K. Maulloo, and D.K.H. Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. Journal of the Operational Research society, 49, 1998.
- [7] 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.
- [8] H. Han, S. Shakkottai, CV Hollot, R. Srikant, and D. Towsley. "Multi-path tcp: a joint congestion control and routing scheme to exploit path diversity in the internet", Trans. on Net., 14, 2006.
- [9] D. Wischik, M. Handly, and C. Raiciu. "Control of multipath tcp and optimization of multipath routing in the

internet", NetCOOP, 2009.

- [10] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handly. "Design, implementation and evaluation of congestion control for multipath tcp", Usenix NSDI, 2011.
- [11] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handly. "Improving datacenter performance and robustness with multipath tcp", ACM Sigcomm, 2011.
- [12] R. Khalili, N. Gast, M. Popovic, U. Upadhyay, and J.-Y. Le Boudec. "Non Pareto-optimality of mptcp: Performance issues and a possible solution", EPFL Technical report. <<http://infoscience.epfl.ch/record/177901>>.
- [13] B. Chen J. Jannotti E. Kohler, R. Morris and M. F. Kaashoek. "The click modular router", Trans. Comput. Syst., 18, August 2000.
- [14] S. Floyd and V. Jacobson. "Random early detection gateways for congestion avoidance", Trans. on Net., 1, August, 1993.

## Authors' Addresses

Ramin Khalili  
EPFL IC ISC LCA2  
Station 14  
CH-1015 Lausanne  
Switzerland

Phone: +41 21 693 5610  
EMail: [ramin.khalili@epfl.ch](mailto:ramin.khalili@epfl.ch)

Nicolas Gast  
EPFL IC ISC LCA2  
Station 14  
CH-1015 Lausanne  
Switzerland

Phone: +41 21 693 1254  
EMail: [nicolas.gast@epfl.ch](mailto:nicolas.gast@epfl.ch)

Miroslav Popovic  
EPFL IC ISC LCA2  
Station 14  
CH-1015 Lausanne  
Switzerland

Phone: +41 21 693 6466  
EMail: [miroslav.popovic@epfl.ch](mailto:miroslav.popovic@epfl.ch)

Jean-Yves Le Boudec  
EPFL IC ISC LCA2  
Station 14  
CH-1015 Lausanne  
Switzerland

Phone: +41 21 693 6631  
EMail: [jean-yves.leboudec@epfl.ch](mailto:jean-yves.leboudec@epfl.ch)



Internet Engineering Task Force  
Internet-Draft  
Intended status: Informational  
Expires: January 15, 2013

R. Winter  
University of Applied Sciences Augsburg  
A. Ripke  
NEC Laboratories Europe  
July 16, 2012

Multipath TCP Support for Single-homed End-systems  
draft-wr-mptcp-single-homed-03

Abstract

Multipath TCP relies on the existence of multiple paths at the end-systems typically provided through different IP addresses obtained by different ISPs. While this scenario is certainly becoming increasingly a reality (e.g. mobile devices), currently most end-systems are single-homed (e.g. desktop PCs in an enterprise). This memo describes mechanisms to make multiple paths available to multipath TCP-capable end-systems that are not available directly at the end-systems but somewhere within the network.

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 15, 2013.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/>)

license-info) in effect on the date of 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.

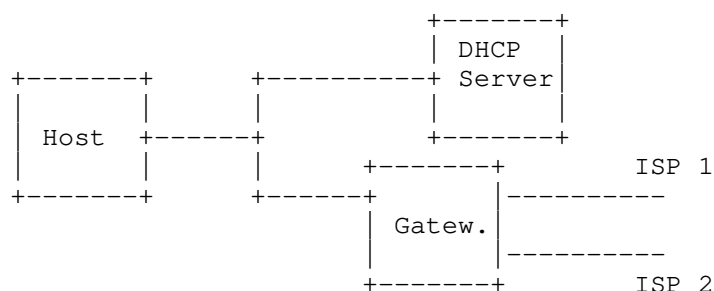
## Table of Contents

1. Introduction . . . . .	2
2. Approaches to Use Multiple Paths in the Network . . . . .	3
2.1. Exposing Multiple Paths Through End-host Auto-configuration	3
2.2. Heuristic Use of Multiple Paths . . . . .	5
3. Other scenarios and extensions . . . . .	6
4. Alternative approaches . . . . .	6
5. Acknowledgements . . . . .	6
6. IANA Considerations . . . . .	7
7. Security Considerations . . . . .	7
8. References . . . . .	7
8.1. Normative References . . . . .	7
8.2. Informative References . . . . .	7
Authors' Addresses . . . . .	7

## 1. Introduction

The IETF is specifying a multipath TCP (MPTCP) architecture and protocol where end-systems operate a modified standard TCP stack which allows packets of the same TCP connection to be sent via different paths to an MPTCP-capable destination ([I-D.ietf-mptcp-multiaddressed], [RFC6182]) where paths are defined by sets of source and destination IP addresses. Using multiple paths has a number of benefits such as an increased reliability of the transport connection and an effect known as resource pooling [resource\_pooling]. Most end-systems today do not have multiple paths/interfaces available in order to make use of multipath TCP, however further within the network multiple paths are the norm rather than the exception. This memo therefore describes ways how these multiple paths in the network could potentially made be available to multipath TCP-capable hosts that are single-homed.

In order to illustrate the general mechanism we make use of a simple reference scenario shown in Figure 1.



The scenario in Figure 1 depicts e.g. a possible SOHO or enterprise setup where a gateway/router is connected to two ISPs and a DHCP server gives out leases to hosts connected to the local network. Note that both, the gateway and the DHCP server could be on the same device (similar to current home gateway implementations).

The host is running a multipath-capable IP stack, however it only has a single interface. The methods described in the following sections will let the host make use of the gateway's two interfaces without requiring modifications to the MPTCP implementation.

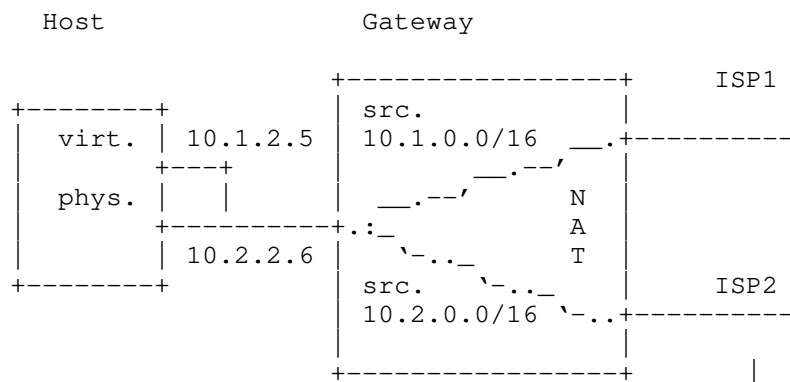
## 2. Approaches to Use Multiple Paths in the Network

All approaches in this document do not require changes to the wire format of MPTCP and both communicating hosts need to be MPTCP-capable. The benefit this approach has is that a) it has no implications on how the MPTCP standard progresses, b) it will hopefully encourage the deployment of MPTCP as the number of scenarios where MPTCP brings benefits vastly expands and c) these approaches do not require complex middle-boxes to implement MPTCP-like functionality in the network as other approaches have suggested before.

### 2.1. Exposing Multiple Paths Through End-host Auto-configuration

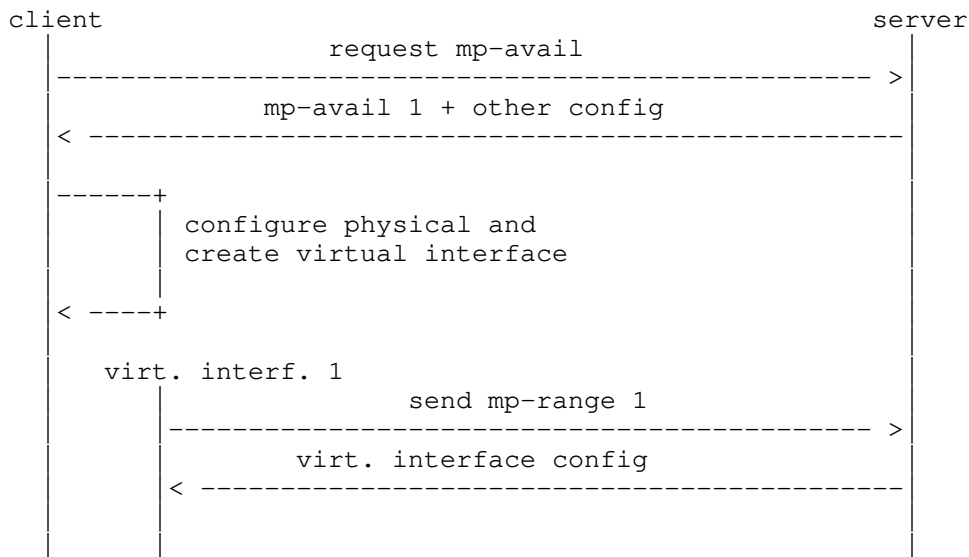
Multipath TCP distinguishes paths by their source and destination IP addresses. Assuming a certain level of path diversity in the Internet, using different source and destination IP addresses for a given subflow of a multipath TCP connection will, with a certain probability, result in different paths taken by packets of different subflows. Even in case subflows share a common bottleneck, the proposed multipath congestion control algorithm [RFC6356] will make sure that multipath TCP will play nicely with regular TCP flows.

In order to not require changes to the TCP implementation, we keep the above assumptions multipath TCP makes, i.e. working with different IP addresses to use different paths. Since the end-system is single-homed, all IP addresses are bound to the same physical interface. In our reference scenario in Figure 1, the host would e.g. receive more than one RFC1918 [RFC1918] private IP address from the DHCP server as depicted in Figure 2.



The gateway that is shown in Figure 2 has received two IP addresses, one from each ISP that it is connected to (ISP1 and ISP2). The NAT that the gateway is implementing needs to "map" each private IP address of the host consistently to a one of the addresses received by the ISPs, i.e. each private IP to a different public IP. Packets sent by the host to the gateway are then routed based on the source address found in the packets as illustrated in the figure. In other words, depending on the source address of the host, the packets will either go through ISP 1 or ISP 2 and TCP will balance the traffic across those two links using its built-in congestion control mechanism.

The way the gateway has received its public IP addresses is not relevant. It could be via DHCP, IPCP or static configuration. In order to configure the host via DHCP, we propose two new DHCP options. The first option "mp-avail" will be sent by single-homed multipath TCP-capable clients in the "Parameter Request List". This will show the DHCP server that the client is multipath-capable. The DHCP server will answer with "mp-avail" and the option value is set to the number of additional interfaces the gateway can offer to the client (in our reference scenario that value would be 1; see Figure 3).



Upon receipt of the "mp-avail" option from the server, the client can create up to  $n$  virtual interfaces, where  $n$  is the option value. Each virtual interface will contact the DHCP server and will include the "mp-range" option. The option value will tell the DHCP server that the client is requesting an IP out of an IP range that the gateway will be forwarding through a different interface.

The above has been implemented using the ISC DHCP server and client version 4.2.1 and the multipath TCP kernel patch 0.5 and a 2.6.36 Linux kernel.

## 2.2. Heuristic Use of Multiple Paths

The auto-configuration mechanism above has the advantage that available paths and information on how to use them are directly sent to the end-host. In other words, there is an explicit signalling of the availability of multiple paths to the end-host. This has the advantage that the host can efficiently use these paths.

This method works well when multiple paths are available close to the end-host and means for auto-configuration are available. But that is not always the case. Another method to use different paths in the network without prior knowledge of their existence is to apply heuristics in order to exploit setups where Equal Cost Multi-path [RFC2991], a widely deployed technology [ECMP\_DEPLOYMENT], or similar per-flow load-balancing algorithms are employed.

The ADD\_ADDR option defined in [I-D.ietf-mptcp-multiaddressed] can be used to advertise the same address but a different port to open another subflow. Additionally, the MP\_JOIN option can also be used to open another subflow with the same IP address and e.g. a different source port given that a different address ID is used. This means there are multiple scenarios possible (e.g. either sender-initiated or receiver-initiated) where single-homed end-hosts can influence the 5-tuple (source and destination IP addresses and port numbers plus protocol number) which is often used as the basis for per-flow load balancing (NOTE: in a future version this document will describe some of these scenarios in more detail). Changing the 5-tuple will only with a certain probability result in using a different path unless the load-balancing algorithm that is used is known to the MPTCP implementation (an assumption we cannot generally make). This means that a number of subflows might end up on the same path. Fortunately, the MPTCP congestion control algorithm will make sure that the collection of subflows on that path will not be more aggressive than a single TCP flow.

### 3. Other scenarios and extensions

The reference scenario is only one conceivable setting. Other scenarios such as DSL broadband customers or mobile phones are conceivable as well. As an example, take the DSL scenario. The home gateway could be provided with multiple IP addresses using extensions to IPCP. The home gateway in turn can then implement the DHCP server and gateway functionality as described before. More scenarios will be described in future versions of this document.

### 4. Alternative approaches

One alternative is that a DHCP server always sends  $n$  offers, where  $n$  is the number of interfaces at the gateway to different ISPs. The client could then accept all or a subset of these offers. This approach seems interesting in environments where there are multiple DHCP servers, one for each ISP connection (think multiple homegateways). However, accepting multiple offers based on a single DHCP request is not standard's compliant behavior. Also, to cater for a scenario that only contains a single DHCP server, server changes are needed in any case. Finally, correct routing is not always guaranteed in these scenarios.

An interesting alternative is the use of ECMP at the gateway for load distribution and let MPTCP use different port numbers for subflows. Assuming that ECMP is available at the gateway, this approach would work fine today. The only drawback of the approach is that it involves a little trial and error to find port numbers that actually hash to different paths used by ECMP [RFC2991].

### 5. Acknowledgements

Part of this work was 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.

## 6. IANA Considerations

Two new DHCP options are required by this version of this document.

## 7. Security Considerations

TBD.

## 8. References

### 8.1. Normative References

- [RFC1918] Rekhter, Y., Moskowitz, R., Karrenberg, D., Groot, G. and E. Lear, "Address Allocation for Private Internets", BCP 5, RFC 1918, February 1996.
- [RFC2991] Thaler, D. and C. Hopps, "Multipath Issues in Unicast and Multicast Next-Hop Selection", RFC 2991, November 2000.

### 8.2. Informative References

- [ECMP\_DEPLOYMENT] Augustin, B., Friedman, T. and R. Teixeira, "Measuring Multipath Routing in the Internet", October 2011, <[http://www.paris-traceroute.net/images/ton\\_2011.pdf](http://www.paris-traceroute.net/images/ton_2011.pdf)>.
- [I-D.ietf-mptcp-multiaddressed] Ford, A., Raiciu, C., Handley, M. and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", Internet-Draft draft-ietf-mptcp-multiaddressed-09, June 2012.
- [RFC6182] Ford, A., Raiciu, C., Handley, M., Barre, S. and J. Iyengar, "Architectural Guidelines for Multipath TCP Development", RFC 6182, March 2011.
- [RFC6356] Raiciu, C., Handley, M. and D. Wischik, "Coupled Congestion Control for Multipath Transport Protocols", RFC 6356, October 2011.
- [resource\_pooling] Wischik, D., Handley, M. and M. Bagnulo Braun, "The Resource Pooling Principle", October 2008, <<http://ccr.sigcomm.org/online/files/p47-handleyA4.pdf>>.

## Authors' Addresses

Rolf Winter  
University of Applied Sciences Augsburg  
An der Hochschule 1  
Augsburg 86161  
Germany

Email: [rolf.winter@hs-augsburg.de](mailto:rolf.winter@hs-augsburg.de)

Andreas Ripke  
NEC Laboratories Europe  
Kurfuersten-Anlage 36  
Heidelberg 69115  
Germany

Email: [andreas.ripke@neclab.eu](mailto:andreas.ripke@neclab.eu)