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

Copyright Notice

Table of Contents

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:

- 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.

- 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.

- 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 stablity 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.

```
                   +-------------------------------+
                   |           Application         |
                   +-------------------------------+
                          ^                 |
               ~~~~~~~~~~~|~Socket Interface|~~~~~~~~~~~
                          |                 v
                  +-------------------------------+
                  |             MPTCP             |
                  + - - - - - - - + - - - - - - - +
                  | Subflow (TCP) | Subflow (TCP) |
                  +-------------------------------+
                  |       IP      |      IP       |
                  +-------------------------------+
```

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 conncetion (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 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 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
TCP_MULTIPATH_ENABLE	o	o	boolean
TCP_MULTIPATH_ADD		o	list of addresses
TCP_MULTIPATH_REMOVE		o	list of addresses
TCP_MULTIPATH_SUBFLOWS	o		list of pairs of addresses
TCP_MULTIPATH_CONNID	o		32-bit integer

Table 1: MPTCP Socket Operations

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

- 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.

- 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.

- 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.

- TCP_MULTIPATH_SUBFLOWS: This value is read-only and SHOULD only be used after connection setup.

- 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:

- sctp_bindx()
- sctp_connectx()
- sctp_getladdrs()
- sctp_getpaddrs()
- sctp_freeladdrs()
- 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 Removel 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.

- Better explanation when TCP_MULTIPATH_REMOVE may be used.

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

- 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.

- Mention possible additional extended API functions for the indication of application characterstics and for backup paths, based on comments received from the community.

- Mentions alternative approaches for avoiding non-MPTCP-capable paths to reduce impact on applications.

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

- Removal of explicit references to "socket options" and getsockopt/ setsockopt.

- Change of TCP_MULTIPATH_BIND to TCP_MULTIPATH_ADD and TCP_MULTIPATH_REMOVE.

- Mention of stability of bandwidth as another potential QoS parameter for the advanced API.

- 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:

- Definition of the behavior of getpeername() and getsockname() when being called by an MPTCP-aware application.

- Discussion of the possiblity that an MPTCP implementation could support the SCTP API, as far as it is applicable to MPTCP.

- Various editorial fixes.

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

- Second half of the document completely restructured

- 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