Network Working Group
Internet-Draft
Intended status: Informational
Expires: December 24, 2018

X. de Foy
M. Perras
InterDigital Communications
U. Chunduri
Huawei USA
K. Nguyen
M. Kibria
K. Ishizu
F. Kojima
NICT

June 22, 2018

Considerations for MPTCP operation in 5G
draft-defoy-mptcp-considerations-for-5g-01

Abstract

This document describes scenarios where the behavior of the 5G mobility management framework is different from earlier cellular generations, and describes how it may benefit from some form of adaptation of MPTCP implementations and protocol aspects in the 5G system. This document also describes how MPTCP may be leveraged in 5G system specifications.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

MPTCP [RFC6824] is being deployed and widely adopted in today's smart devices, which typically have multiple network interfaces such as Cellular and Wifi. It provides reliability, bandwidth aggregation capability, and handover efficiency.

This document describes scenarios where the behavior of the 5G mobility management framework is different from earlier cellular generations, and describes how it may benefit from some form of adaptation of MPTCP implementations and protocol aspects in the 5G system (see Section 1.1). This document also describes how MPTCP may be leveraged in 5G system specifications (see Section 1.2).

1.1. Need for Adapting MPTCP to 5G Session and Service Continuity

With regards to 5G session and service continuity mechanisms, MPTCP stack behavior (including path manager and scheduling) should be updated to achieve optimal performance:

- o Proposed 5G MPTCP behavior is described in Section 2.2, Section 2.3 and Section 2.4.

- o To enable this behavior, MPTCP will need additional information about local IP addresses: (1) its session continuity service type and (2) a notification that a new IP address has been provided for service continuity. This is described in Section 2.1.

- o Remote MPTCP peers may need access to the IP address SCS type (through MPTCP signaling or otherwise). This is discussed in Section 2.5.

1.2. Opportunity for Leveraging MPTCP in 5G Dual Connectivity Feature

With regards to dual connectivity, MPTCP can be closely integrated with the 5G stack to avoid duplicating its feature in 5G, as discussed in Section 3. As a summary:

- o MPTCP should be aware of the presence of multiple DC radio links, which may be exposed as a single or distinct network interfaces/IP addresses.

- o MPTCP should optimally partition traffic or duplicate a data flow over DC links, depending on the application’s need, network policy and conditions.

2. Impact of 5G Session and Service Continuity on MPTCP

One of the goals of 5G systems, as outlined in [_3GPP.23.501], is to enable low latency services and access to local data networks where mobility anchors can be deployed close to devices, thereby satisfying use cases with stringent transmission delay and high reliability. Mobility in 4G networks, as described at the architecture level in [_3GPP.23.401], was based on a central mobility solution that made it difficult to relocate mobility anchors closer to the end user. In contrast, 5G uses a distributed mobility solution based on multiple anchors providing different IP addresses as the device moves from one area to another.

The base scenario in this section is: a 5G device connected to a single-homed server is in an area with no usable Wifi coverage. An application using MPTCP sends traffic over a single subflow, over the cellular air interface. Then, as the device moves, the 5G device reacts to mobility events. Additionally, we briefly discuss cases where the device switches from a Wifi to a cellular backup connection, and other cases where both MPTCP peers are 5G mobile devices.

In 5G, every unit of a network connectivity service (PDU session) has a type which can be IP (IPv4 or IPv6), Ethernet or unstructured. While session continuity is supported for all types, we will focus on PDU sessions of IP type primarily. Different PDU sessions will typically correspond to distinct network interfaces on the device (though this is not explicit in the standard, and some implementations may possibly behave differently).

In 4G networks, session continuity is enabled by anchoring a PDN Connection (as PDU Sessions are referred to in 4G networks) to a P-GW which allocates an IP address to the mobile device: PDN connection and IP address allocation are maintained as long as the device remains attached to the network, even when the device moves around. In 5G, different types of session continuity can be provided, and are indicated by a "Session and Service Continuity" (SSC) mode value of 1, 2 or 3 (defined in [_3GPP.23.501] section 5.6.9). Every PDU session is associated with a single SSC mode, which cannot be changed on this PDU session. The following sub-sections will study how 5G handles each SSC mode, and potential effects on MPTCP.

2.1. New Inputs for MPTCP stacks

IP Address Session Continuity Service Type

The "session continuity service type" (SCS type) of an IP address allocated by the network can be used as input by MPTCP implementations. It has been defined for on-demand mobility management [I-D.ietf-dmm-ondemand-mobility], as:

FIXED IP address: valid for a very long time, for session continuity and IP address reachability.

SESSION_LASTING IP address: valid for the lifetime of an IP session, even if the mobility host moves.

NON_PERSISTENT IP address: which does not provide session continuity nor IP address reachability.

GRACEFUL_REPLACEMENT IP address: similar to a non-persistent address, but adding a limited graceful period for the transition from one address to another.

This information needs to be conveyed to the device by the network that allocates the address: for example, as described in [I-D.feng-dmm-ra-prefixtype], the SCS type of IP address may be conveyed through router advertisements. The MPTCP stack may obtain the SCS type of a local IP address using a programmatic API. This information does not directly tell which SSC mode the application is using, nevertheless it is sufficient for MPTCP to appropriately and non-ambiguously decide how to handle new IP addresses in the context of SSC in 5G.

Notification of a New IP Address from Session Continuity

As in non-5G cases, MPTCP starts handling the initial socket created by the application. The initial connection attempt triggers the creation of the first PDU session (or the reuse of an existing one), associated with a network interface and an IP address on this interface. Later on, the network decides to use a new anchor, e.g., when a mobility event occurs or when a Local Data Network becomes available. The network then triggers the modification of the existing PDU session, or creation of a new PDU session, and ultimately a new IP address is provided to the 5G device, respectively on the same or a new network interface.

The 5G stack on the device is aware of the relationship between the old and new IP addresses, and can therefore notify the MPTCP stack. Such a notification may include both the old and new IP addresses , which ensures that the MPTCP stack has all necessary information (for example, the MPTCP stack may then obtain the SCS type of the new IP address, and the validity lifetime associated with the old IP address, if it is still up).

2.2. SSC mode 1

In SSC mode 1 the same network anchor is kept regardless of device location. An application running on the device will therefore be able to keep using the same IP address on the same interface.

Additionally, in SSC mode 1, the network may decide to add and remove, dynamically, additional network anchors (and therefore IP addresses) to the PDU session, while always keeping the initial one.

PROPOSED MPTCP BEHAVIOR SPECIFICATION:

These steps are applicable when the initial IP address returned by the network stack is FIXED or SESSION_LASTING.

MPTCP should not close all subflows originated from this original IP address at any point during the session, since this IP address is the only one that is guaranteed, under normal circumstances, to be maintained over time for this application.

At any time during the session, a new IP address of SCS type NON_PERSISTENT may become available. MPTCP may create new subflows for the application, using this IP address (this IP address is likely to provide shorter path subflows, but may disappear at any time).

2.3. SSC mode 2

SSC mode 2 has a break-before-make behavior. When the device leaves the service area of its first network anchor, the network stops using it and starts using a new second network anchor closer to the device. (Such service areas may have a highly variable size depending on network deployments.) On the device, this can result in the currently used network interface being brought down, and after a short time a new network interface being brought up. The time between these 2 events is not standardized and implementation dependent.

Break-before-make within cellular technology

When MPTCP is active on cellular only, this break-before-make behavior is similar to the existing break-before-make capability usually used in cellular/Wifi offload (section 3.1.3 of [RFC6897] and section 2.2 of [RFC8041]). A similar MPTCP behavior is therefore needed: wait for a given time for a new IP address to be configured. As per [RFC6897], to benefit from this MPTCP resilience feature, the application should not request using a specific network interface.

Cellular and Wifi

Additionally, when Wifi is active and cellular is used as backup, MPTCP implementations should also support this break-before-make behavior to maintain a usable backup IP address on cellular. In rare cases where a switch-to-cellular backup would be needed during a break-before-make transition on cellular, MPTCP's existing break-before-make capability can ensure MPTCP waits for a new cellular-facing IP address to be available.

PROPOSED MPTCP BEHAVIOR SPECIFICATION:

These steps are applicable when the initial IP address returned by the network stack is NON_PERSISTENT.

At any point in time, the current NON_PERSISTENT IP address may be taken down by the network stack. The MPTCP stack should wait for another NON_PERSISTENT IP address to be made available by the network stack. If such an address is made available within a given time limit, the MPTCP stack should create new subflows using this new address (effectively following the existing break-before-make behavior present in MPTCP).

Additionally, if an initial backup IP address is a NON_PERSISTENT address, the MPTCP stack should consider any subsequent NON_PERSISTENT IP address as a backup IP address in replacement of the initial NON_PERSISTENT address.

2.4. SSC mode 3

SSC mode 3 has a make-before-break behavior. When the device leaves the service area of its first network anchor, the network selects a second network anchor closer to the device, and either creates a new PDU session (i.e. new IP address on new network interface) or share the existing PDU session (i.e. new IP address on same network interface). The first network anchor keeps being used for a given time period, which is communicated to the device by the network using the "valid lifetime" field of a prefix information option in a router advertisement ([RFC4861], [RFC4862]). 5G does not mandate a specific range for this valid lifetime. The first/older IP address should not be used to create any new traffic ([RFC4862] section 5.5.4). In some implementations, the network (SMF) may decide to release the first network anchor as soon as it stops carrying traffic.

There is no limit set by the 5G standard for the number of concurrently used network anchors. We expect that in usual cases the first network anchor will be released before a third network anchor starts being used. Nevertheless, to our knowledge nothing prevents a 5G system deployment to allow a third network anchor to be selected while the first one is still in use.

On the 5G device, when using SSC mode 3, mobility will therefore result in a new IP address being configured, either on the same network interface initially used, or on a different interface. In general, an application will see a single cellular-facing IP address, and during transient phase it will see 2 IP addresses (with a possibility for more than 2 concurrent IP addresses on some 5G system implementations). In cases where the server is single-homed and the Wifi interface is down, and assuming a full-mesh path manager policy, there will be in general one subflow, and from time to time, temporarily 2 subflows (or more on some 5G systems). In cases where two mobile 5G devices are communicating with each other over MPTCP and with the same assumptions on Wifi and path manager policy, there will be in general one subflow, and from time to time, temporarily 2 or even more rarely 4 subflows (again, possibly more on some 5G systems).

MPTCP must create new subflows when a new IP address on a same or a new cellular-facing network interface becomes available to the application. MPTCP may keep using the first subflow during a transient phase. Here are some considerations related to this transient phase:

o When compared with simply waiting for the first IP address to be brought down, ramping down usage of the first subflow will not incur inefficiencies from resending lost segments. This may especially help low-latency applications by avoiding throughput drop.

o Assuming a lowest-rtt-first scheduling policy is used, after the initial TCP slow start, the shortest path subflow should typically carry the most traffic. Ramping down should ideally start after the initial slow start is over.

o To make sure the ramping down completes before the interface is brought down by the network, the MPTCP stack should be aware of how long will the first network anchor be kept in use, e.g. through configuration or communication with the local 5G stack.

o Ramping down and closing flows on the first network anchor as soon as possible will help recycling network resources more rapidly. This is especially true in cases where more than 2 network anchors may be used concurrently.

o There may be some level of contention between subflows during the transient phase, since they share the same air interface, and especially if they share the same PDU session and QoS marking. The shortest path subflow may therefore not reach its full capacity during the transient phase.

o Additionally, the shortest subflow must not be closed during the transient phase (even if it is less efficient for some reason), to avoid losing all connectivity at the end of the transient phase. To avoid this issue, the MPTCP stack could for example follow a policy not to close any subflow created using the latest IP address, during the transient period (in SSC mode 3).

In cases where cellular is used for backup, there is a possibility that the switch to using backup occurs during a transient phase. To support this case, MPTCP should keep creating and releasing subflows as described above, even when cellular subflows are used as backup, to ensure that the backup is always usable. When a backup event occurs during a transient phase, MPTCP should use the subflows associated with the most recent cellular-facing IP address, i.e. corresponding to the latest/closest network anchor.

PROPOSED MPTCP BEHAVIOR SPECIFICATION:

These steps are applicable when the initial IP address returned by the network stack is GRACEFUL_REPLACEMENT.

> At any point in time, a new GRACEFUL_REPLACEMENT IP address may be made available by the network stack. The MPTCP stack must create new subflows using this new address, gracefully transfer traffic to these new subflow(s), and close subflow(s) using the previous GRACEFUL_REPLACEMENT IP address before its scheduled closing (known by obtaining the valid lifetime of the IP address from the operating system).

> Additionally, if an initial backup IP address is a GRACEFUL_REPLACEMENT address, the MPTCP stack should consider any subsequent GRACEFUL_REPLACEMENT IP address as the new backup IP address, in replacement of the first GRACEFUL_REPLACEMENT IP address.

2.5. Behavior of Remote Peer

The SCS type of an IP address is only available locally, which limits to only one peer the MPTCP behavior specified in sections Section 2.2, Section 2.3 and Section 2.4. This can potentially cause problems:

> A remote may for example close subflows that should be maintained e.g. closing all subflows to a SESSION_LASTING IP address or to the latest GRACEFUL_REPLACEMENT address.

> A remote may also not behave optimally in some cases, e.g. because it cannot distinguish between a transition from GRACEFUL_REPLACEMENT to GRACEFUL_REPLACEMENT, from a temporary addition of a NON_PERSISTENT address to a SESSION_LASTING address.

There are two possible ways to handle this: either explicitly transmit the SCS type over MPTCP signalling, or use heuristics on the remote peer (e.g. never close a subflow created by a peer, mirror behavior of peer, etc.). In the explicit solution, a MPTCP peer provides the SCS type of IP addresses to its remote peer, using existing or new options (mp_capable, add_addr, mp_join, or an experimental option). When compared to using heuristics, transmitting the SCS type can lead to a less complex and more explicit implementation, which can help better handling unusual or error cases, and adapt to future evolution of 5G or other mobility management systems.

3. MPTCP with 5G Dual Connectivity

One of the key features of 5G [_3GPP.23.501] is dual connectivity (DC). With DC, a 5G device can be served by two different base stations. DC may play an essential role in leveraging the benefit of 5G new radio, especially in the evolving architecture with the coexistence of 4G and 5G radios.

On a 5G device with DC, an application is able to send data to the destination (e.g., a single-home server) through multiple radio links, over one or more PDU sessions. Some PDU sessions may be over a single radio link, while others may have flows over each radio link. Therefore, in a first case, DC can be made visible to applications through different IP addresses, while in a second case, DC can be used by different flows terminated at the same IP address on the device.

In any of those cases, the issues of out of order delivery and diverse latency values need to be supported in DC. However, such reliable communication scenarios have not been addressed in the current DC architecture. Based on the design history of DC in earlier systems, the 5G system will need to incorporate features to support robustness/reliability (e.g., backup and duplication), that will likely result in added complexity. Meanwhile, similar issues have been solved in MPTCP (e.g., two types of sequences at subflow and connection levels for out of order delivery, etc.). MPTCP hence could be leveraged in those scenarios. On the other hand, to satisfy new QoS requirements of 5G applications as well as to benefit the most from DC, 5G devices are expected to include advanced algorithms that control data transmissions over each radio link optimally. For example, those algorithms could aim to minimize overall latency or satisfy latency requirements of applications. Hence, the 5G device needs to dynamically select the most suitable path for a given radio condition. Additionally, algorithms for shifting, based on congestion, ongoing traffic between transmissions over radio links are also necessary.

MPTCP, which includes path manager, scheduler, and congestion control functions, shows a lot of potential to address the issues mentioned above. MPTCP could, therefore, be integrated with DC and the 5G protocol stack, as an alternative to developing 5G-specific solutions. As part of this integration, the MPTCP stack should be aware of the presence of multiple radio links, whether they are exposed using multiple IP addresses or hidden under a single IP address. MPTCP’s scheduler should optimally partition traffic or duplicate a data flow over different links, depending on the application’s need, network policy, and conditions.

4. IANA Considerations

This document requests no IANA actions.

5. Security Considerations

No new security considerations are identified at this time.

6. Acknowledgements

Thanks to following people for contributing, reviewing or providing feedback: Debashish Purkayastha, Akbar Rahman, Ulises Olvera-Hernandez, Sri Gundavelli.

7. Informative References

[_3GPP.23.401]
3GPP, "General Packet Radio Service (GPRS) enhancements for Evolved Universal Terrestrial Radio Access Network (E-UTRAN) access", 3GPP TS 23.401 15.3.0, 3 2018, <http://www.3gpp.org/ftp/Specs/html-info/23401.htm>.

[_3GPP.23.501]
3GPP, "System Architecture for the 5G System", 3GPP TS 23.501 15.14.0, 3 2018, <http://www.3gpp.org/ftp/Specs/html-info/23501.htm>.

[I-D.feng-dmm-ra-prefixtype]
Feng, W. and D. Moses, "Router Advertisement Extensions for On-Demand Mobility", draft-feng-dmm-ra-prefixtype-02 (work in progress), March 2018.

[I-D.ietf-dmm-ondemand-mobility]
Yegin, A., Moses, D., Kweon, K., Lee, J., Park, J., and S. Jeon, "On Demand Mobility Management", draft-ietf-dmm-ondemand-mobility-14 (work in progress), March 2018.

[RFC4861] Narten, T., Nordmark, E., Simpson, W., and H. Soliman, "Neighbor Discovery for IP version 6 (IPv6)", RFC 4861, DOI 10.17487/RFC4861, September 2007, <https://www.rfc-editor.org/info/rfc4861>.

[RFC4862] Thomson, S., Narten, T., and T. Jinmei, "IPv6 Stateless Address Autoconfiguration", RFC 4862, DOI 10.17487/RFC4862, September 2007, <https://www.rfc-editor.org/info/rfc4862>.

[RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <https://www.rfc-editor.org/info/rfc6824>.

[RFC6897] Scharf, M. and A. Ford, "Multipath TCP (MPTCP) Application Interface Considerations", RFC 6897, DOI 10.17487/RFC6897, March 2013, <https://www.rfc-editor.org/info/rfc6897>.

[RFC8041] Bonaventure, O., Paasch, C., and G. Detal, "Use Cases and Operational Experience with Multipath TCP", RFC 8041, DOI 10.17487/RFC8041, January 2017, <https://www.rfc-editor.org/info/rfc8041>.

Authors' Addresses

Xavier de Foy
InterDigital Communications, LLC
1000 Sherbrooke West
Montreal
Canada

Email: Xavier.Defoy@InterDigital.com

Michelle Perras
InterDigital Communications, LLC
Montreal
Canada

Email: Michelle.Perras@InterDigital.com

Uma Chunduri
Huawei USA
2330 Central Expressway
Santa Clara, CA 95050
USA

Email: uma.chunduri@huawei.com

Kien Nguyen
National Institute of Information and Communications Technology
YRP Center No.1 Building 7F, 3-4 Hikarinooka, Yokosuka
Kanagawa 239-0847
Japan

Email: kienng@nict.go.jp

Mirza Golam Kibria
National Institute of Information and Communications Technology
YRP Center No.1 Building 7F, 3-4 Hikarinooka, Yokosuka
Kanagawa 239-0847
Japan

Email: mirza.kibria@nict.go.jp

Kentaro Ishizu
National Institute of Information and Communications Technology
YRP Center No.1 Building 7F, 3-4 Hikarinooka, Yokosuka
Kanagawa 239-0847
Japan

Email: ishidu@nict.go.jp

Fumihide Kojima
National Institute of Information and Communications Technology
YRP Center No.1 Building 7F, 3-4 Hikarinooka, Yokosuka
Kanagawa 239-0847
Japan

Email: f-kojima@nict.go.jp

Internet Engineering Task Force
Internet-Draft
Obsoletes: 6824 (if approved)
Intended status: Standards Track
Expires: December 10, 2019

A. Ford
Pexip
C. Raiciu
U. Politechnica of Bucharest
M. Handley
U. College London
O. Bonaventure
U. catholique de Louvain
C. Paasch
Apple, Inc.
June 8, 2019

TCP Extensions for Multipath Operation with Multiple Addresses
draft-ietf-mptcp-rfc6824bis-18

Abstract

TCP/IP communication is currently restricted to a single path per connection, yet multiple paths often exist between peers. The simultaneous use of these multiple paths for a TCP/IP session would improve resource usage within the network and, thus, improve user experience through higher throughput and improved resilience to network failure.

Multipath TCP provides the ability to simultaneously use multiple paths between peers. This document presents a set of extensions to traditional TCP to support multipath operation. The protocol offers the same type of service to applications as TCP (i.e., reliable bytestream), and it provides the components necessary to establish and use multiple TCP flows across potentially disjoint paths.

This document specifies v1 of Multipath TCP, obsoleting v0 as specified in RFC6824, through clarifications and modifications primarily driven by deployment experience.

Status of This Memo

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 10, 2019.

Copyright Notice

Table of Contents

1. Introduction

Multipath TCP (MPTCP) is a set of extensions to regular TCP [RFC0793] to provide a Multipath TCP [RFC6182] 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 [RFC6182], 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 [RFC6356] presents a safe congestion control algorithm for coupling the behavior of the multiple paths in order to "do no harm" to other network users.

o Application considerations [RFC6897] discusses what impact MPTCP will have on applications, what applications will want to do with MPTCP, and as a consequence of these factors, what API extensions an MPTCP implementation should present.

This document is an update to, and obsoletes, the v0 specification of Multipath TCP (RFC6824). This document specifies MPTCP v1, which is not backward compatible with MPTCP v0. This document additionally defines version negotiation procedures for implementations that support both versions.

1.1. Design Assumptions

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

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

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

To simplify the design, we assume that the presence of multiple addresses at a host is sufficient to indicate the existence of multiple paths. These paths need not be entirely disjoint: they may share one or many routers between them. Even in such a situation, making use of multiple paths is beneficial, improving resource utilization and resilience to a subset of node failures. The congestion control algorithms defined in [RFC6356] ensure this does not act detrimentally. Furthermore, there may be some scenarios where different TCP ports on a single host can provide disjoint paths (such as through certain Equal-Cost Multipath (ECMP) implementations

[RFC2992]), and so the MPTCP design also supports the use of ports in path identifiers.

There are three aspects to the backwards-compatibility listed above (discussed in more detail in [RFC6182]):

External Constraints: The protocol must function through the vast majority of existing middleboxes such as NATs, firewalls, and proxies, and as such must resemble existing TCP as far as possible on the wire. Furthermore, the protocol must not assume the segments it sends on the wire arrive unmodified at the destination: they may be split or coalesced; TCP options may be removed or duplicated.

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

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

The complementary application considerations document [RFC6897] discusses the necessary features of an API to provide backwards-compatibility, as well as API extensions to convey the behavior of MPTCP at a level of control and information equivalent to that available with regular, single-path TCP.

Further discussion of the design constraints and associated design decisions are given in the MPTCP Architecture document [RFC6182] and in [howhard].

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 [RFC6897].

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

Figure 1: Comparison of Standard TCP and MPTCP Protocol Stacks

1.3. Terminology

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

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

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

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

Data-level: The payload data is nominally transferred over a connection, which in turn is transported over subflows. Thus, the term "data-level" is synonymous with "connection level", in contrast to "subflow-level", which refers to properties of an individual subflow.

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

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

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

1.4. MPTCP Concept

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

- To a non-MPTCP-aware application, MPTCP will behave the same as normal TCP. Extended APIs could provide additional control to MPTCP-aware applications [RFC6897]. An application begins by opening a TCP socket in the normal way. MPTCP signaling and operation are handled by the MPTCP implementation.

- 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.

- 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.

- 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 or B2.

- 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.

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

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

```
               Host A                               Host B
      ------------------------             ------------------------
      Address A1    Address A2             Address B1    Address B2
      ----------    ----------             ----------    ----------
          |             |                      |             |
          |     (initial connection setup)     |             |
          |----------------------------------->|             |
          |<-----------------------------------|             |
          |             |                      |             |
          |            (additional subflow setup)            |
          |             |--------------------->|             |
          |             |<---------------------|             |
          |             |                      |             |
          |             |                      |             |
```

Figure 2: Example MPTCP Usage Scenario

1.5. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Operation Overview

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

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

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

2.1. Initiating an MPTCP Connection

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

```
Host A                                  Host B
------                                  ------
MP_CAPABLE                ->
[flags]
                          <-            MP_CAPABLE
                                        [B's key, flags]
ACK + MP_CAPABLE (+ data) ->
[A's key, B's key, flags, (data-level details)]
```

Retransmission of the ACK + MP_CAPABLE can occur if it is not known if it has been received. The following diagrams show all possible exchanges for the initial subflow setup to ensure this reliability.

```
      Host A (with data to send immediately)  Host B
      ------                                  ------
      MP_CAPABLE                ->
      [flags]
                                <-            MP_CAPABLE
                                              [B's key, flags]
      ACK + MP_CAPABLE + data   ->
      [A's key, B's key, flags, data-level details]


      Host A (with data to send later)        Host B
      ------                                  ------
      MP_CAPABLE                ->
      [flags]
                                <-            MP_CAPABLE
                                              [B's key, flags]
      ACK + MP_CAPABLE          ->
      [A's key, B's key, flags]

      ACK + MP_CAPABLE + data   ->
      [A's key, B's key, flags, data-level details]


      Host A                                  Host B (sending first)
      ------                                  ------
      MP_CAPABLE                ->
      [flags]
                                <-            MP_CAPABLE
                                              [B's key, flags]
      ACK + MP_CAPABLE          ->
      [A's key, B's key, flags]

                                <-            ACK + DSS + data
                                              [data-level details]
```

2.2. Associating a New Subflow with an Existing MPTCP Connection

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

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

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

                         <-                ACK
```

2.3. Informing the Other Host about Another Potential Address

The set of IP addresses associated to a multihomed host may change during the lifetime of an MPTCP connection. MPTCP supports the addition and removal of addresses on a host both implicitly and explicitly. If Host A has established a subflow starting at address/port pair IP#-A1 and wants to open a second subflow starting at address/port pair 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/port pair (IP#-A2). Host B may later send an MP_JOIN to this new address. The ADD_ADDR option contains a HMAC to authenticate the address as having been sent from the originator of the connection. The receiver of this option echoes it back to the client to indicate successful receipt. Further details are in Section 3.4.1.

```
    Host A                                   Host B
    ------                                   ------
    ADD_ADDR                   ->
    [Echo-flag=0,
     IP#-A2,
     IP#-A2's Address ID,
     HMAC of IP#-A2]

                               <-          ADD_ADDR
                                          [Echo-flag=1,
                                           IP#-A2,
                                           IP#-A2's Address ID,
                                           HMAC of IP#-A2]
```

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

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

2.4. Data Transfer Using MPTCP

To ensure reliable, in-order delivery of data over subflows that may appear and disappear at any time, MPTCP uses a 64-bit data sequence number (DSN) to number all data sent over the MPTCP connection. Each subflow has its own 32-bit sequence number space, utilising the regular TCP sequence number header, 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 (DSS) carries the Data Sequence Mapping. The Data Sequence Mapping consists of the subflow sequence number, data sequence number, and length for which this mapping is valid. This option can also carry a connection-level acknowledgment (the "Data ACK") for the received DSN.

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

Further details are in Section 3.3.

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

2.5. Requesting a Change in a Path's Priority

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

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

2.6. Closing an MPTCP Connection

When a host wants to close an existing subflow, but not the whole connection, it can initiate a regular TCP FIN/ACK exchange.

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

```
Host A                               Host B
------                               ------
DSS                       ->
[Data FIN]
                          <-         DSS
                                     [Data ACK]
```

There is an additional method of connection closure, referred to as "Fast Close", which is analogous to closing a single-path TCP connection with a RST signal. The MP_FASTCLOSE signal is used to indicate to the peer that the connection will be abruptly closed and no data will be accepted anymore. This can be used on an ACK (ensuring reliability of the signal), or a RST (which is not). Both examples are shown in the following diagrams. Further details are in Section 3.5.

```
   Host A                               Host B
   ------                               ------
   ACK + MP_FASTCLOSE          ->
   [B's key]

   [RST on all other subfluws] ->

                               <-       [RST on all subflows]


   Host A                               Host B
   ------                               ------
   RST + MP_FASTCLOSE          ->
   [B's key] [on all subflows]

                               <-       [RST on all subflows]
```

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 receiver of the SYN 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. This is discussed in Section 3.7.

- o To address the threats identified in [RFC6181], the following steps are taken: keys are sent in the clear in the MP_CAPABLE messages; MP_JOIN messages are secured with HMAC-SHA256 ([RFC2104], [RFC6234]) using those keys; and standard TCP validity checks are made on the other messages (ensuring sequence numbers are in-window [RFC5961]). Residual threats to MPTCP v0 were identified in [RFC7430], and those affecting the protocol (i.e. modification to ADD_ADDR) have been incorporated in this document. Further discussion of security can be found in Section 5.

3. MPTCP Protocol

This section describes the operation of the MPTCP protocol, and is subdivided into sections for each key part of the protocol operation.

All MPTCP operations are signaled using optional TCP header fields. A single TCP option number ("Kind") has been assigned by IANA for MPTCP (see Section 8), and then individual messages will be determined by a "subtype", the values of which are also stored in an IANA registry (and are also listed in Section 8). As with all TCP options, the Length field is specified in bytes, and includes the 2 bytes of Kind and Length.

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

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

Figure 3: MPTCP Option Format

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

The remaining options, however, are signals that do not need to be on a specific packet, such as those for signaling additional addresses. Whilst an implementation may desire to send MPTCP options as soon as possible, it may not be possible to combine all desired options (both those for MPTCP and for regular TCP, such as SACK (selective acknowledgment) [RFC2018]) 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 [RFC5681] in regular TCP. Therefore, an MPTCP implementation receiving a duplicate ACK that contains an MPTCP option MUST NOT treat it as a signal of congestion. Additionally, an MPTCP implementation SHOULD NOT send more than two duplicate ACKs in a row for the purposes of sending MPTCP options alone, in order to ensure no middleboxes misinterpret this as a sign of congestion.

Furthermore, standard TCP validity checks (such as ensuring the sequence number and acknowledgment number are within window) MUST be undertaken before processing any MPTCP signals, as described in [RFC5961], and initial subflow sequence numbers SHOULD be generated according to the recommendations in [RFC6528].

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) MPTCP option (Figure 4). This option declares its sender is capable of performing Multipath TCP and wishes to do so on this particular connection.

The MP_CAPABLE exchange in this specification (v1) is different to that specified in v0. If a host supports multiple versions of MPTCP, the sender of the MP_CAPABLE option SHOULD signal the highest version number it supports. In return, in its MP_CAPABLE option, the receiver will signal the version number it wishes to use, which MUST be equal to or lower than the version number indicated in the initial MP_CAPABLE. There is a caveat though with respect to this version negotiation with old listeners that only support v0. A listener that supports v0 expects that the MP_CAPABLE option in the SYN-segment includes the initiator’s key. If the initiator however already upgraded to v1, it won’t include the key in the SYN-segment. Thus, the listener will ignore the MP_CAPABLE of this SYN-segment and reply with a SYN/ACK that does not include an MP_CAPABLE. The initiator MAY choose to immediately fall back to TCP or MAY choose to attempt a connection using MPTCP v0 (if the initiator supports v0), in order to discover whether the listener supports the earlier version of MPTCP. In general a MPTCP v0 connection is likely to be preferred to a TCP one, however in a particular deployment scenario it may be known that the listener is unlikely to support MPTCPv0 and so the initiator may prefer not to attempt a v0 connection. An initiator MAY cache information for a peer about what version of MPTCP it supports if any, and use this information for future connection attempts.

The MP_CAPABLE option is variable-length, with different fields included depending on which packet the option is used on. The full MP_CAPABLE option is shown in Figure 4.

```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-------+---------------+
|     Kind      |    Length     |Subtype|Version|A|B|C|D|E|F|G|H|
+---------------+---------------+-------+-------+---------------+
|                   Option Sender's Key (64 bits)               |
|                      (if option Length > 4)                   |
|                                                               |
+---------------------------------------------------------------+
|                  Option Receiver's Key (64 bits)              |
|                      (if option Length > 12)                  |
|                                                               |
+-------------------------------+-------------------------------+
|  Data-Level Length (16 bits)  |  Checksum (16 bits, optional) |
+-------------------------------+-------------------------------+
```

Figure 4: Multipath Capable (MP_CAPABLE) Option

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

- SYN (A->B): only the first four octets (Length = 4).

- SYN/ACK (B->A): B's Key for this connection (Length = 12).

- ACK (no data) (A->B): A's Key followed by B's Key (Length = 20).

- ACK (with first data) (A->B): A's Key followed by B's Key followed by Data-Level Length, and optional Checksum (Length = 22 or 24).

The contents of the option is determined by the SYN and ACK flags of the packet, along with 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).

The initial SYN, containing just the MP_CAPABLE header, is used to define the version of MPTCP being requested, as well as exchanging flags to negotiate connection features, described later.

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

Upon reception of the initial SYN-segment, a stateful server generates a random key and replies with a SYN/ACK. The key's method of generation is implementation specific. The key MUST be hard to guess, and it MUST be unique for the sending host across all its current MPTCP connections. Recommendations for generating random numbers for use in keys are given in [RFC4086]. Connections will be indexed at each host by the token (a one-way hash of the key). Therefore, an implementation will require a mapping from each token to the corresponding connection, and in turn to the keys for the connection.

There is a risk that two different keys will hash to the same token. The risk of hash collisions is usually small, unless the host is handling many tens of thousands of connections. Therefore, an implementation SHOULD check its list of connection tokens to ensure there is no collision before sending its key, and if there is, then it should generate a new key. This would, however, be costly for a server with thousands of connections. The subflow handshake mechanism (Section 3.2) will ensure that new subflows only join the correct connection, however, through the cryptographic handshake, as well as checking the connection tokens in both directions, and ensuring sequence numbers are in-window. So in the worst case if there was a token collision, the new subflow would not succeed, but the MPTCP connection would continue to provide a regular TCP service.

Since key generation is implementation-specific, there is no requirement that they be simply random numbers. An implementation is free to exchange cryptographic material out-of-band and generate these keys from this, in order to provide additional mechanisms by which to verify the identity of the communicating entities. For example, an implementation could choose to link its MPTCP keys to those used in higher-layer TLS or SSH connections.

If the server behaves in a stateless manner, it has to generate its own key in a verifiable fashion. This verifiable way of generating the key can be done by using a hash of the 4-tuple, sequence number and a local secret (similar to what is done for the TCP-sequence number [RFC4987]). It will thus be able to verify whether it is indeed the originator of the key echoed back in the later MP_CAPABLE option. As for a stateful server, the tokens SHOULD be checked for uniqueness, however if uniqueness is not met, and there is no way to generate an alternative verifiable key, then the connection MUST fall back to using regular TCP by not sending a MP_CAPABLE in the SYN/ACK.

The ACK carries both A's key and B's key. This is the first time that A's key is seen on the wire, although it is expected that A will have generated a key locally before the initial SYN. The echoing of B's key allows B to operate statelessly, as described above. Therefore, A's key must be delivered reliably to B, and in order to do this, the transmission of this packet must be made reliable.

If B has data to send first, then the reliable delivery of the ACK+MP_CAPABLE can be inferred by the receipt of this data with a MPTCP Data Sequence Signal (DSS) option (Section 3.3). If, however, A wishes to send data first, it has two options to ensure the reliable delivery of the ACK+MP_CAPABLE. If it immediately has data to send, then the third ACK (with data) would also contain an MP_CAPABLE option with additional data parameters (the Data-Level Length and optional Checksum as shown in Figure 4). If A does not immediately have data to send, it MUST include the MP_CAPABLE on the third ACK, but without the additional data parameters. When A does have data to send, it must repeat the sending of the MP_CAPABLE option from the third ACK, with additional data parameters. This MP_CAPABLE option is in place of the DSS, and simply specifies the data-level length of the payload, and the checksum (if the use of checksums is negotiated). This is the minimal data required to establish a MPTCP connection - it allows validation of the payload, and given it is the first data, the Initial Data Sequence Number (IDSN) is also known (as it is generated from the key, as described below). Conveying the keys on the first data packet allows the TCP reliability mechanisms to ensure the packet is successfully delivered. The receiver will acknowledge this data at the connection level with a Data ACK, as if a DSS option has been received.

There could be situations where both A and B attempt to transmit initial data at the same time. For example, if A did not initially have data to send, but then needed to transmit data before it had received anything from B, it would use a MP_CAPABLE option with data parameters (since it would not know if the MP_CAPABLE on the ACK was received). In such a situation, B may also have transmitted data with a DSS option, but it had not yet been received at A. Therefore, B has received data with a MP_CAPABLE mapping after it has sent data with a DSS option. To ensure these situations can be handled, it follows that the data parameters in a MP_CAPABLE are semantically equivalent to those in a DSS option and can be used interchangeably. Similar situations could occur when the MP_CAPABLE with data is lost and retransmitted. Furthermore, in the case of TCP Segmentation Offloading, the MP_CAPABLE with data parameters may be duplicated across multiple packets, and implementations must also be able to cope with duplicate MP_CAPABLE mappings as well as duplicate DSS mappings.

Additionally, the MP_CAPABLE exchange allows the safe passage of MPTCP options on SYN packets to be determined. If any of these options are dropped, MPTCP will gracefully fall back to regular single-path TCP, as documented in Section 3.7. If at any point in the handshake either party thinks the MPTCP negotiation is compromised, for example by a middlebox corrupting the TCP options, or unexpected ACK numbers being present, the host MUST stop using MPTCP and no longer include MPTCP options in future TCP packets. The other host will then also fall back to regular TCP using the fall back mechanism. Note that new subflows MUST NOT be established (using the process documented in Section 3.2) until a Data Sequence Signal (DSS) option has been successfully received across the path (as documented in Section 3.3).

Like all MPTCP options, the MP_CAPABLE option starts with the Kind and Length to specify the TCP-option kind and its length. Followed by that is the MP_CAPABLE option. The first 4 bits of the first octet in the MP_CAPABLE option (Figure 4) define the MPTCP option subtype (see Section 8; for MP_CAPABLE, this is 0x0), and the remaining 4 bits of this octet specify the MPTCP version in use (for this specification, this is 1).

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

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

B: The second bit, labeled "B", is an extensibility flag, and MUST be set to 0 for current implementations. This will be used for an extensibility mechanism in a future specification, and the impact of this flag will be defined at a later date. It is expected, but not mandated, that this flag would be used as part of an alternative security mechanism that does not require a full version upgrade of the protocol, but does require redefining some elements of the handshake. If receiving a message with the 'B' flag set to 1, and this is not understood, then the MP_CAPABLE in this SYN MUST be silently ignored, which triggers a fallback to regular TCP; the sender is expected to retry with a format compatible with this legacy specification. Note that the length of the MP_CAPABLE option, and the meanings of bits "D" through "H", may be altered by setting B=1.

C: The third bit, labeled "C", is set to "1" to indicate that the sender of this option will not accept additional MPTCP subflows to the source address and port, and therefore the receiver MUST NOT try to open any additional subflows towards this address and port. This is an efficiency improvement for situations where the sender knows a restriction is in place, for example if the sender is behind a strict NAT, or operating behind a legacy Layer 4 load balancer.

D through H: The remaining bits, labeled "D" through "H", are used for crypto algorithm negotiation. In this specification only the rightmost bit, labeled "H", is assigned. Bit "H" indicates the use of HMAC-SHA256 (as defined in Section 3.2). An implementation that only supports this method MUST set bit "H" to 1, and bits "D" through "G" to 0.

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

The selection of the authentication algorithm also impacts the algorithm used to generate the token and the Initial Data Sequence Number (IDSN). In this specification, with only the SHA-256 algorithm (bit "H") specified and selected, the token MUST be a truncated (most significant 32 bits) SHA-256 hash ([RFC6234]) of the key. A different, 64-bit truncation (the least significant 64 bits) of the SHA-256 hash of the key MUST be used as the IDSN. Note that the key MUST be hashed in network byte order. Also note that the "least significant" bits MUST be the rightmost bits of the SHA-256 digest, as per [RFC6234]. Future specifications of the use of the crypto bits may choose to specify different algorithms for token and IDSN generation.

Both the crypto and checksum bits negotiate capabilities in similar ways. For the Checksum Required bit (labeled "A"), if either host requires the use of checksums, checksums MUST be used. In other words, the only way for checksums not to be used is if both hosts in their SYNs set A=0. This decision is confirmed by the setting of the "A" bit in the third packet (the ACK) of the handshake. For example, if the initiator sets A=0 in the SYN, but the responder sets A=1 in the SYN/ACK, checksums MUST be used in both directions, and the initiator will set A=1 in the ACK. The decision whether to use checksums will be stored by an implementation in a per-connection binary state variable. If A=1 is received by a host that does not want to use checksums, it MUST fall back to regular TCP by ignoring the MP_CAPABLE option as if it was invalid.

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

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

If a SYN contains an MP_CAPABLE option but the SYN/ACK does not, it is assumed that sender of the SYN/ACK is not multipath capable; thus, the MPTCP session MUST operate as a regular, single-path TCP. If a SYN does not contain a MP_CAPABLE option, the SYN/ACK MUST NOT contain one in response. If the third packet (the ACK) does not contain the MP_CAPABLE option, then the session MUST fall back to operating as a regular, single-path TCP. This is to maintain compatibility with middleboxes on the path that drop some or all TCP options. Note that an implementation MAY choose to attempt sending MPTCP options more than one time before making this decision to operate as regular TCP (see Section 3.9).

If the SYN packets are unacknowledged, it is up to local policy to decide how to respond. It is expected that a sender will eventually fall back to single-path TCP (i.e., without the MP_CAPABLE option) in order to work around middleboxes that may drop packets with unknown options; however, the number of multipath-capable attempts that are made first will be up to local policy. It is possible that MPTCP and non-MPTCP SYNs could get reordered in the network. Therefore, the final state is inferred from the presence or absence of the MP_CAPABLE option in the third packet of the TCP handshake. If this option is not present, the connection SHOULD fall back to regular TCP, as documented in Section 3.7.

The initial data sequence number on an MPTCP connection is generated from the key. The algorithm for IDSN generation is also determined from the negotiated authentication algorithm. In this specification, with only the SHA-256 algorithm specified and selected, the IDSN of a host MUST be the least significant 64 bits of the SHA-256 hash of its key, i.e., IDSN-A = Hash(Key-A) and IDSN-B = Hash(Key-B). This deterministic generation of the IDSN allows a receiver to ensure that there are no gaps in sequence space at the start of the connection. The SYN with MP_CAPABLE occupies the first octet of data sequence space, although this does not need to be acknowledged at the connection level until the first data is sent (see Section 3.3).

3.2. Starting a New Subflow

Once an MPTCP connection has begun with the MP_CAPABLE exchange, further subflows can be added to the connection. Hosts have knowledge of their own address(es), and can become aware of the other host's addresses through signaling exchanges as described in Section 3.4. Using this knowledge, a host can initiate a new subflow over a currently unused pair of addresses. It is permitted for either host in a connection to initiate the creation of a new subflow, but it is expected that this will normally be the original connection initiator (see Section 3.9 for heuristics).

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

This section specifies the behavior of MP_JOIN using the HMAC-SHA256 algorithm. An MP_JOIN option is present in the SYN, SYN/ACK, and ACK of the three-way handshake, although in each case with a different format.

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

The token is used to identify the MPTCP connection and is a cryptographic hash of the receiver's key, as exchanged in the initial MP_CAPABLE handshake (Section 3.1). In this specification, the tokens presented in this option are generated by the SHA-256 [RFC6234] algorithm, truncated to the most significant 32 bits. The token included in the MP_JOIN option is the token that the receiver of the packet uses to identify this connection; i.e., Host A will send Token-B (which is generated from Key-B). Note that the hash generation algorithm can be overridden by the choice of cryptographic handshake algorithm, as defined in Section 3.1.

The MP_JOIN SYN sends not only the token (which is static for a connection) but also random numbers (nonces) that are used to prevent replay attacks on the authentication method. Recommendations for the generation of random numbers for this purpose are given in [RFC4086].

The MP_JOIN option includes an "Address ID". This is an identifier generated by the sender of the option, used to identify the source address of this packet, even if the IP header has been changed in transit by a middlebox. The numeric value of this field is generated by the sender and must map uniquely to a source IP address for the sending host. The Address ID allows address removal (Section 3.4.2) without needing to know what the source address at the receiver is, thus allowing address removal through NATs. The Address ID also allows correlation between new subflow setup attempts and address signaling (Section 3.4.1), to prevent setting up duplicate subflows on the same path, if an MP_JOIN and ADD_ADDR are sent at the same time.

The Address IDs of the subflow used in the initial SYN exchange of the first subflow in the connection are implicit, and have the value zero. A host MUST store the mappings between Address IDs and addresses both for itself and the remote host. An implementation will also need to know which local and remote Address IDs are associated with which established subflows, for when addresses are removed from a local or remote host.

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

```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-----+-+---------------+
|     Kind      |  Length = 12  |Subtype|(rsv)|B|   Address ID  |
+---------------+---------------+-------+-----+-+---------------+
|                   Receiver's Token (32 bits)                  |
+---------------------------------------------------------------+
|                Sender's Random Number (32 bits)               |
+---------------------------------------------------------------+
```

Figure 5: Join Connection (MP_JOIN) Option (for Initial SYN)

When receiving a SYN with an MP_JOIN option that contains a valid token for an existing MPTCP connection, the recipient SHOULD respond with a SYN/ACK also containing an MP_JOIN option containing a random number and a truncated (leftmost 64 bits) Hash-based Message Authentication Code (HMAC). This version of the option is shown in Figure 6. If the token is unknown, or the host wants to refuse subflow establishment (for example, due to a limit on the number of subflows it will permit), the receiver will send back a reset (RST) signal, analogous to an unknown port in TCP, containing a MP_TCPRST option (Section 3.6) with a "MPTCP specific error" reason code. Although calculating an HMAC requires cryptographic operations, it is believed that the 32-bit token in the MP_JOIN SYN gives sufficient protection against blind state exhaustion attacks; therefore, there is no need to provide mechanisms to allow a responder to operate statelessly at the MP_JOIN stage.

An HMAC is sent by both hosts -- by the initiator (Host A) in the third packet (the ACK) and by the responder (Host B) in the second packet (the SYN/ACK). Doing the HMAC exchange at this stage allows both hosts to have first exchanged random data (in the first two SYN packets) that is used as the "message". This specification defines that HMAC as defined in [RFC2104] is used, along with the SHA-256 hash algorithm [RFC6234], and that the output is truncated to the leftmost 160 bits (20 octets). Due to option space limitations, the HMAC included in the SYN/ACK is truncated to the leftmost 64 bits, but this is acceptable since random numbers are used; thus, an attacker only has one chance to correctly guess the HMAC that matches the random number previously sent by the peer (if the HMAC is incorrect, the TCP connection is closed, so a new MP_JOIN negotiation with a new random number is required).

The initiator's authentication information is sent in its first ACK (the third packet of the handshake), as shown in Figure 7. This data needs to be sent reliably, since it is the only time this HMAC is sent; therefore, receipt of this packet MUST trigger a regular TCP ACK in response, and the packet MUST be retransmitted if this ACK is not received. In other words, sending the ACK/MP_JOIN packet places the subflow in the PRE_ESTABLISHED state, and it moves to the ESTABLISHED state only on receipt of an ACK from the receiver. It is not permitted to send data while in the PRE_ESTABLISHED state. The reserved bits in this option MUST be set to zero by the sender.

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

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

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

```
                         1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +---------------+---------------+-------+-----------------------+
    |     Kind      |  Length = 24  |Subtype|      (reserved)       |
    +---------------+---------------+-------+-----------------------+
    |                                                               |
    |                                                               |
    |              Sender's Truncated HMAC (160 bits)               |
    |                                                               |
    |                                                               |
    +---------------------------------------------------------------+
```

Figure 7: Join Connection (MP_JOIN) Option (for Third ACK)

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

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

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

Figure 8: Example Use of MPTCP Authentication

If the token received at Host B is unknown or local policy prohibits the acceptance of the new subflow, the recipient MUST respond with a TCP RST for the subflow. If appropriate, a MP_TCPRST option with a "Administratively prohibited" reason code (Section 3.6) should be included.

If the token is accepted at Host B, but the HMAC returned to Host A does not match the one expected, Host A MUST close the subflow with a TCP RST. In this, and all following cases of sending a RST in this section, the sender SHOULD send a MP_TCPRST option (Section 3.6) on this RST packet with the reason code for a "MPTCP specific error".

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

If the HMACs are verified as correct, then both hosts have verified 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 TCP 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 listener on the relevant port, it will respond with a RST in the normal way. If in response to a SYN with an MP_JOIN option, a SYN/ACK is received without the MP_JOIN option (either since it was stripped on the return path, or it was stripped on the outgoing path but 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 3.9 for heuristics); no explicit application-level accept calls or bind calls are required to open additional subflows. To associate a new subflow with an existing connection, the token supplied in the subflow's SYN exchange is used for demultiplexing. This then binds the 5-tuple of the TCP subflow to the local token of the connection. A consequence is that it is possible to allow any port pairs to be used for a connection.

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

3.3. General MPTCP Operation

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

The data sequence mapping and the Data ACK are signaled in the Data Sequence Signal (DSS) option (Figure 9). Either or both can be signaled in one DSS, depending 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.

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

Figure 9: Data Sequence Signal (DSS) Option

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

- A = Data ACK present
- a = Data ACK is 8 octets (if not set, Data ACK is 4 octets)
- M = Data Sequence Number (DSN), Subflow Sequence Number (SSN), Data-Level Length, and Checksum (if negotiated) present
- m = Data sequence number is 8 octets (if not set, DSN is 4 octets)

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

The 'F' flag indicates "Data FIN". If present, this means that this mapping covers the final data from the sender. This is the connection-level equivalent to the FIN flag in single-path TCP. A connection is not closed unless there has been a Data FIN exchange, a MP_FASTCLOSE (Section 3.5) message, or an implementation-specific, connection-level send timeout. The purpose of the Data FIN and the interactions between this flag, the subflow-level FIN flag, and the data sequence mapping are described in Section 3.3.3. The remaining reserved bits MUST be set to zero by an implementation of this specification.

Note that the checksum is only present in this option if the use of MPTCP checksumming has been negotiated at the MP_CAPABLE handshake (see Section 3.1). The presence of the checksum can be inferred from the length of the option. If a checksum is present, but its use had not been negotiated in the MP_CAPABLE handshake, the receiver MUST close the subflow with a RST as it not behaving as negotiated. If a checksum is not present when its use has been negotiated, the receiver MUST close the subflow with a RST as it is considered broken. In both cases, this RST SHOULD be accompanied with a MP_TCPRST option (Section 3.6) with the reason code for a "MPTCP specific error".

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 [RFC2018] is used at the subflow level to improve efficiency.

The data sequence mapping specifies a mapping from subflow sequence space to data sequence space. This is expressed in terms of starting sequence numbers for the subflow and the data level, and a length of bytes for which this mapping is valid. This explicit mapping for a range of data was chosen rather than per-packet signaling to assist with compatibility with situations where TCP/IP segmentation or coalescing is undertaken separately from the stack that is generating the data flow (e.g., through the use of TCP segmentation offloading on network interface cards, or by middleboxes such as performance enhancing proxies). It also allows a single mapping to cover many packets, which may be useful in bulk transfer situations.

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

The data sequence number is specified as an absolute value, whereas the subflow sequence numbering is relative (the SYN at the start of the subflow has relative subflow sequence number 0). This is to allow middleboxes to change the initial sequence number of a subflow, such as firewalls that undertake Initial Sequence Number (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 3.7, since MPTCP can no longer reliably know the subflow sequence space at the receiver to build data sequence mappings. Without checksumming enabled, corrupt data may be delivered to the application if a middlebox alters segment boundaries, alters content, or does not deliver all segments covered by a data sequence mapping. It is therefore RECOMMENDED to use checksumming unless it is known the network path contains no such devices.

The checksum algorithm used is the standard TCP checksum [RFC0793], operating over the data covered by this mapping, along with a pseudo-header as shown in Figure 10.

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

Figure 10: Pseudo-Header for DSS Checksum

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

Note that checksumming relies on the TCP subflow containing contiguous data; therefore, a TCP subflow MUST NOT use the Urgent

Pointer to interrupt an existing mapping. Further note, however, that if Urgent data is received on a subflow, it SHOULD be mapped to the data sequence space and delivered to the application analogous to Urgent data in regular TCP.

To avoid possible deadlock scenarios, subflow-level processing should be undertaken separately from that at connection level. Therefore, even if a mapping does not exist from the subflow space to the data-level space, the data SHOULD still be ACKed at the subflow (if it is in-window). This data cannot, however, be acknowledged at the data level (Section 3.3.2) because its data sequence numbers are unknown. Implementations MAY hold onto such unmapped data for a short while in the expectation that a mapping will arrive shortly. Such unmapped data cannot be counted as being within the connection level receive window because this is relative to the data sequence numbers, so if the receiver runs out of memory to hold this data, it will have to be discarded. If a mapping for that subflow-level sequence space does not arrive within a receive window of data, that subflow SHOULD be treated as broken, closed with a RST, and any unmapped data silently discarded.

Data sequence numbers are always 64-bit quantities, and MUST be maintained as such in implementations. If a connection is progressing at a slow rate, so protection against wrapped sequence numbers is not required, then an implementation MAY 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 be able to receive and process both 64-bit or 32-bit sequence number values, but it is not required that an implementation is able to send both.

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) [RFC7323]. The lengths of the DSNs used in these values (which may be different) are declared with flags in the DSS option. Implementations MUST accept a 32-bit DSN and implicitly promote it to a 64-bit quantity by incrementing the upper 32 bits of sequence number each time the lower 32 bits wrap. A sanity check MUST be implemented to ensure that a wrap occurs at an expected time (e.g., the sequence number jumps from a very high number to a very low number) and is not triggered by out-of-order packets.

As with the standard TCP sequence number, the data sequence number should not start at zero, but at a random value to make blind session hijacking harder. This specification requires setting the initial data sequence number (IDSN) of each host to the least significant 64 bits of the SHA-256 hash of the host's key, as described in Section 3.1. This is required also in order for the receiver to know what the expected IDSN is, and thus determine if any initial connection-level packets are missing; this is particularly relevant if two subflows start transmitting simultaneously.

A data sequence mapping does not need to be included in every MPTCP packet, as long as the subflow sequence space in that packet is covered by a mapping known at the receiver. This can be used to reduce overhead in cases where the mapping is known in advance; one such case is when there is a single subflow between the hosts, another is when segments of data are scheduled in larger than packet-sized chunks.

An "infinite" mapping can be used to fall back to regular TCP by mapping the subflow-level data to the connection-level data for the remainder of the connection (see Section 3.7). This is achieved by setting the Data-Level Length field of the DSS option to the reserved value of 0. The checksum, in such a case, will also be set to zero.

3.3.2. Data Acknowledgments

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

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

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

An MPTCP sender MUST NOT free data from the send buffer until it has been acknowledged by both a Data ACK received on any subflow and at the subflow level by all subflows on which the data was sent. The former condition ensures liveness of the connection and the latter condition ensures liveness and self-consistence of a subflow when data needs to be retransmitted. Note, however, that if some data needs to be retransmitted multiple times over a subflow, there is a risk of blocking the sending window. In this case, the MPTCP sender can decide to terminate the subflow that is behaving badly by sending a RST, using an appropriate MP_TCPRST (Section 3.6) error code.

The Data ACK MAY be included in all segments; however, optimizations SHOULD be considered in more advanced implementations, where the Data ACK is present in segments only when the Data ACK value advances, and this behavior MUST be treated as valid. This behavior ensures the sender buffer is freed, while reducing overhead when the data transfer is unidirectional.

3.3.3. Closing a Connection

In regular TCP, a FIN announces the receiver that the sender has no more data to send. In order to allow subflows to operate independently and to keep the appearance of TCP over the wire, a FIN in MPTCP only affects the subflow on which it is sent. This allows nodes to exercise considerable freedom over which paths are in use at any one time. The semantics of a FIN remain as for regular TCP; i.e., it is not until both sides have ACKed each other's FINs that the subflow is fully closed.

When an application calls close() on a socket, this indicates that it has no more data to send; for regular TCP, this would result in a FIN on the connection. For MPTCP, an equivalent mechanism is needed, and this is referred to as the DATA_FIN.

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

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

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

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

Once a DATA_FIN has been acknowledged, all remaining subflows MUST be closed with standard FIN exchanges. Both hosts SHOULD send FINs on all subflows, as a courtesy to allow middleboxes to clean up state even if an individual subflow has failed. It is also encouraged to reduce the timeouts (Maximum Segment Lifetime) on subflows at end hosts after receiving a DATA_FIN. In particular, any subflows where there is still outstanding data queued (which has been retransmitted on other subflows in order to get the DATA_FIN acknowledged) MAY be closed with a RST with MP_TCPRST (Section 3.6) error code for "too much outstanding data".

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 D). This permits "break-before-make" scenarios where connectivity is lost on all subflows before a new one can be re-established.

3.3.4. Receiver Considerations

Regular TCP advertises a receive window in each packet, telling the sender how much data the receiver is willing to accept past the cumulative ack. The receive window is used to implement flow control, throttling down fast senders when receivers cannot keep up.

MPTCP also uses a unique receive window, shared between the subflows. The idea is to allow any subflow to send data as long as the receiver is willing to accept it. The alternative, maintaining per subflow receive windows, could end up stalling some subflows while others would not use up their window.

The receive window is relative to the DATA_ACK. As in TCP, a receiver MUST NOT shrink the right edge of the receive window (i.e., DATA_ACK + receive window). The receiver will use the data sequence number to tell if a packet should be accepted at the connection level.

When deciding to accept packets at subflow level, regular TCP checks the sequence number in the packet against the allowed receive window. With multipath, such a check is done using only the connection-level window. A sanity check SHOULD be performed at subflow level to ensure that the subflow and mapped sequence numbers meet the following test: SSN - SUBFLOW_ACK <= DSN - DATA_ACK, where SSN is the subflow sequence number of the received packet and SUBFLOW_ACK is the RCV.NXT (next expected sequence number) of the subflow (with the equivalent connection-level definitions for DSN and DATA_ACK).

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

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

3.3.5. Sender Considerations

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

MPTCP uses a single receive window across all subflows, and if the receive window was guaranteed to be unchanged end-to-end, a host could always read the most recent receive window value. However, some classes of middleboxes may alter the TCP-level receive window. Typically, these will shrink the offered window, although for short periods of time it may be possible for the window to be larger (however, note that this would not continue for long periods since ultimately the middlebox must keep up with delivering data to the receiver). Therefore, if receive window sizes differ on multiple subflows, when sending data MPTCP SHOULD take the largest of the most recent window sizes as the one to use in calculations. This rule is implicit in the requirement not to reduce the right edge of the window.

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

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

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

3.3.6. Reliability and Retransmissions

The data sequence mapping allows senders to resend data with the same data sequence number on a different subflow. When doing this, a host MUST still retransmit the original data on the original subflow, in order to preserve the subflow integrity (middleboxes could replay old data, and/or could reject holes in subflows), and a receiver will ignore these retransmissions. While this is clearly suboptimal, for compatibility reasons this is sensible behavior. Optimizations could be negotiated in future versions of this protocol. Note also that this property would also permit a sender to always send the same data, with the same data sequence number, on multiple subflows, if desired for reliability reasons.

This protocol specification does not mandate any mechanisms for handling retransmissions, and much will be dependent upon local policy (as discussed in Section 3.3.8). One can imagine aggressive connection-level retransmissions policies where every packet lost at subflow level is retransmitted on a different subflow (hence, wasting bandwidth but possibly reducing application-to-application delays), or conservative retransmission policies where connection-level retransmits are only used after a few subflow-level retransmission timeouts occur.

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

The sender MUST keep data in its send buffer as long as the data has not been acknowledged at both connection level and on all subflows on which it has been sent. In this way, the sender can always retransmit the data if needed, on the same subflow or on a different one. A special case is when a subflow fails: the sender will typically resend the data on other working subflows after a timeout, and will keep trying to retransmit the data on the failed subflow too. The sender will declare the subflow failed after a predefined upper bound on retransmissions is reached (which MAY be lower than the usual TCP limits of the Maximum Segment Life), or on the receipt of an ICMP error, and only then delete the outstanding data segments.

If multiple retransmissions are triggered that indicate that a subflow performs badly, this MAY lead to a host resetting the subflow with a RST. However, additional research is required to understand the heuristics of how and when to reset underperforming subflows.

For example, a highly asymmetric path may be misdiagnosed as underperforming. A RST for this purpose SHOULD be accompanied with an "Unacceptable performance" MP_TCPRST option (Section 3.6).

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 [RFC6356]; the algorithm does not achieve perfect resource pooling but is "safe" in that it is readily deployable in the current Internet. By this, we mean that it does not take up more capacity on any one path than if it was a single path flow using only that route, so this ensures fair coexistence with single-path TCP at shared bottlenecks.

It is foreseeable that different congestion controllers will be implemented for MPTCP, each aiming to achieve different properties in the resource pooling/fairness/stability design space, as well as those for achieving different properties in quality of service, reliability, and resilience.

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

3.3.8. Subflow Policy

Within a local MPTCP implementation, a host may use any local policy it wishes to decide how to share the traffic to be sent over the available paths.

In the typical use case, where the goal is to maximize throughput, all available paths will be used simultaneously for data transfer, using coupled congestion control as described in [RFC6356]. 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 [RFC6897].

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.

To enable this, 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.

Another use of the MP_PRIO option is to set the 'B' flag on a subflow to cleanly retire its use before closing it and removing it with REMOVE_ADDR Section 3.4.2, for example to support make-before-break session continuity, where new subflows are added before the previously used ones are closed.

```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-----+-+
|     Kind      |     Length    |Subtype|(rsv)|B|
+---------------+---------------+-------+-----+-+
```

Figure 11: Change Subflow Priority (MP_PRIO) Option

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

3.4. Address Knowledge Exchange (Path Management)

We use the term "path management" to refer to the exchange of information about additional paths between hosts, which in this design is managed by multiple addresses at hosts. For more detail of the architectural thinking behind this design, see the MPTCP Architecture document [RFC6182].

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:

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

- 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.

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

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

3.4.1. Address Advertisement

The Add Address (ADD_ADDR) MPTCP option announces additional addresses (and optionally, ports) on which a host can be reached (Figure 12). 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, e.g. [RFC5961], before acting upon it.

Every address has an Address ID that can be used for uniquely identifying the address within a connection for address removal. The Address ID 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 for the sender of the option (within the scope of the connection), but the mechanism for allocating such IDs is implementation specific.

All address IDs learned via either MP_JOIN or ADD_ADDR SHOULD be stored by the receiver in a data structure that gathers all the Address ID to address mappings for a connection (identified by a token pair). In this way, there is a stored mapping between Address ID, observed source address, and token pair for future processing of control information for a connection. Note that an implementation MAY discard incoming address advertisements at will, for example, for avoiding updating mapping state, or because advertised addresses are of no use to it (for example, IPv6 addresses when it has IPv4 only). Therefore, a host MUST treat address advertisements as soft state, and it MAY choose to refresh advertisements periodically. Note also that an implementation MAY choose to cache these address advertisements even if they are not currently relevant but may be relevant in the future, such as IPv4 addresses when IPv6 connectivity is available but IPv4 is awaiting DHCP.

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

The 2 octets that specify the TCP port number to use are optional and their presence 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 3.9.

The Truncated HMAC present in this Option is the rightmost 64 bits of an HMAC, negotiated and calculated in the same way as for MP_JOIN as described in Section 3.2. For this specification of MPTCP, as there is only one hash algorithm option specified, this will be HMAC as defined in [RFC2104], using the SHA-256 hash algorithm [RFC6234]. In the same way as for MP_JOIN, the key for the HMAC algorithm, in the case of the message transmitted by Host A, will be Key-A followed by Key-B, and in the case of Host B, Key-B followed by Key-A. These are the keys that were exchanged in the original MP_CAPABLE handshake. The message for the HMAC is the Address ID, IP Address, and Port which precede the HMAC in the ADD_ADDR option. If the port is not present in the ADD_ADDR option, the HMAC message will nevertheless include two octets of value zero. The rationale for the HMAC is to prevent unauthorized entities from injecting ADD_ADDR signals in an attempt to hijack a connection. Note that additionally the presence of this HMAC prevents the address being changed in flight unless the key is known by an intermediary. If a host receives an ADD_ADDR option for which it cannot validate the HMAC, it SHOULD silently ignore the option.

A set of four flags are present after the subtype and before the Address ID. Only the rightmost bit - labelled 'E' - is assigned in this specification. The other bits are currently unassigned and MUST be set to zero by a sender and MUST be ignored by the receiver.

The 'E' flag exists to provide reliability for this option. Because this option will often be sent on pure ACKs, there is no guarantee of reliability. Therefore, a receiver receiving a fresh ADD_ADDR option (where E=0), will send the same option back to the sender, but not including the HMAC, and with E=1, to indicate receipt. The lack of this echo can be used by the initial ADD_ADDR sender to retransmit the ADD_ADDR according to local policy.

```
                         1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +---------------+---------------+-------+-------+---------------+
    |     Kind      |     Length    |Subtype|(rsv)|E|  Address ID   |
    +---------------+---------------+-------+-------+---------------+
    |          Address (IPv4 - 4 octets / IPv6 - 16 octets)         |
    +-------------------------------+-------------------------------+
    |   Port (2 octets, optional)   |                               |
    +-------------------------------+                               |
    |                Truncated HMAC (8 octets, if E=0)              |
    |                               +-------------------------------+
    |                               |
    +-------------------------------+
```

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 [RFC1918]. 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 advertise such addresses. The MP_JOIN handshake to create a new subflow (Section 3.2) provides mechanisms to minimize security risks. The MP_JOIN message contains a 32-bit token that uniquely identifies the connection to the receiving host. If the token is unknown, the host will return with a RST. In the unlikely event that the token is valid at the receiving host, subflow setup will continue, but the HMAC exchange must occur for authentication. This will fail, and will provide sufficient protection against two unconnected hosts accidentally setting up a new subflow upon the signal of a private address. Further security considerations around the issue of ADD_ADDR messages that accidentally misdirect, or maliciously direct, new MP_JOIN attempts are discussed in Section 5.

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

A host can therefore send an ADD_ADDR message with an already assigned Address ID, but the Address MUST be the same as previously assigned to this Address ID. A new ADD_ADDR may have the same, or different, port number. If the port number is different, the receiving host SHOULD try to set up a new subflow to this new address/port combination.

A host wishing to replace an existing Address ID MUST first remove the existing one (Section 3.4.2).

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 [RFC5681], 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, or an IPv6 address is no longer preferred), the affected host SHOULD announce this so that the peer can remove subflows related to this address. Even if an address is not in use by a MPTCP connection, if it has been previously announced, an implementation SHOULD announce its removal. A host MAY also choose to announce that a valid IP address should not be used any longer, for example for make-before-break session continuity.

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 [RFC1122] on the path, and if a response is received the path SHOULD NOT be removed. If the path is found to still be alive, the receiving host SHOULD no longer use the specified address for future connections, but it is the responsibility of the host which sent the REMOVE_ADDR to shut down the subflow. The requesting host MAY also use MP_PRIO (Section 3.3.8) to request a path is no longer used, before removal. Typical TCP validity tests on the subflow (e.g., ensuring sequence and ACK numbers are correct) MUST also be undertaken. An implementation can use indications of these test failures as part of intrusion detection or error logging.

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

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

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

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

Figure 13: Remove Address (REMOVE_ADDR) Option

3.5. Fast Close

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

MP_FASTCLOSE is used to indicate to the peer that the connection will be abruptly closed and no data will be accepted anymore. The reasons for triggering an MP_FASTCLOSE are implementation specific. Regular TCP does not allow sending a RST while the connection is in a synchronized state [RFC0793]. Nevertheless, implementations allow the sending of a RST in this state, if, for example, the operating system is running out of resources. In these cases, MPTCP should send the MP_FASTCLOSE. This option is illustrated in Figure 14.

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

Figure 14: Fast Close (MP_FASTCLOSE) Option

If Host A wants to force the closure of an MPTCP connection, it has two different options:

- Option A (ACK) : 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.

- Option R (RST) : Host A sends a RST containing the MP_FASTCLOSE option on all subflows, containing the key of Host B as declared in the initial connection handshake. Host A can tear the subflows and the connection down immediately.

If host A decides to force the closure by using Option A and sending an ACK with the MP_FASTCLOSE option, the connection shall proceed as follows:

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

- 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.

- If Host A does not receive a TCP RST in reply to its MP_FASTCLOSE after one retransmission timeout (RTO) (the RTO of the subflow where the MP_FASTCLOSE has been sent), it SHOULD retransmit the MP_FASTCLOSE. The number of retransmissions SHOULD be limited to avoid this connection from being retained for a long time, but this limit is implementation specific. A RECOMMENDED number is 3. If no TCP RST is received in response, Host A SHOULD send a TCP RST with the MP_FASTCLOSE option itself when it releases state in order to clear any remaining state at middleboxes.

If however host A decides to force the closure by using Option R and sending a RST with the MP_FASTCLOSE option, Host B will act as follows: Upon receipt of a RST with MP_FASTCLOSE, containing the valid key, Host B tears down all subflows by sending a TCP RST. Host B can now close the whole MPTCP connection (it transitions directly to CLOSED state).

3.6. Subflow Reset

An implementation of MPTCP may also need to send a regular TCP RST to force the closure of a subflow. A host sends a TCP RST in order to close a subflow or reject an attempt to open a subflow (MP_JOIN). In order to inform the receiving host why a subflow is being closed or rejected, the TCP RST packet MAY include the MP_TCPRST Option. The host MAY use this information to decide, for example, whether it tries to re-establish the subflow immediately, later, or never.

```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-----------------------+
|     Kind      |    Length     |Subtype|U|V|W|T|    Reason     |
+---------------+---------------+-------+-----------------------+
```

Figure 15: TCP RST Reason (MP_TCPRST) Option

The MP_TCPRST option contains a reason code that allows the sender of the option to provide more information about the reason for the termination of the subflow. Using 12 bits of option space, the first four bits are reserved for flags (only one of which is currently defined), and the remaining octet is used to express a reason code for this subflow termination, from which a receiver MAY infer information about the usability of this path.

The "T" flag is used by the sender to indicate whether the error condition that is reported is Transient (T bit set to 1) or Permanent (T bit set to 0). If the error condition is considered to be Transient by the sender of the RST segment, the recipient of this segment MAY try to reestablish a subflow for this connection over the failed path. The time at which a receiver may try to re-establish this is implementation-specific, but SHOULD take into account the properties of the failure defined by the following reason code. If the error condition is considered to be permanent, the receiver of the RST segment SHOULD NOT try to reestablish a subflow for this connection over this path. The "U", "V" and "W" flags are not defined by this specification and are reserved for future use. An implementation of this specification MUST set these flags to 0, and a receiver MUST ignore them.

The "Reason" code is an 8-bit field that indicates the reason for the termination of the subflow. The following codes are defined in this document:

- o Unspecified error (code 0x0). This is the default error implying the subflow is no longer available. The presence of this option shows that the RST was generated by a MPTCP-aware device.

- o MPTCP specific error (code 0x01). An error has been detected in the processing of MPTCP options. This is the usual reason code to return in the cases where a RST is being sent to close a subflow for reasons of an invalid response.

- o Lack of resources (code 0x02). This code indicates that the sending host does not have enough resources to support the terminated subflow.

- o Administratively prohibited (code 0x03). This code indicates that the requested subflow is prohibited by the policies of the sending host.

- o Too much outstanding data (code 0x04). This code indicates that there is an excessive amount of data that need to be transmitted over the terminated subflow while having already been acknowledged over one or more other subflows. This may occur if a path has been unavailable for a short period and it is more efficient to reset and start again than it is to retransmit the queued data.

- o Unacceptable performance (code 0x05). This code indicates that the performance of this subflow was too low compared to the other subflows of this Multipath TCP connection.

- o Middlebox interference (code 0x06). Middlebox interference has been detected over this subflow making MPTCP signaling invalid. For example, this may be sent if the checksum does not validate.

3.7. Fallback

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

MPTCP options on a path, and the modification of payload data. If such an event occurs, it is necessary to "fall back" to the previous, safe operation. This may be either falling back to regular TCP or removing a problematic subflow.

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

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

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

If, however, an ACK is received for data (not just for the SYN) without a DSS option containing a Data ACK, the sender determines the path is not MPTCP capable. In the case of this occurring on an additional subflow (i.e., one started with MP_JOIN), the host MUST close the subflow with a RST, which SHOULD contain a MP_TCPRST option (Section 3.6) with a "Middlebox interference" reason code.

In the case of such an ACK being received on the first subflow (i.e., that started with MP_CAPABLE), before any additional subflows are added, the implementation 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 (to inform the other end in case the path drops options in one direction only), and then revert to sending data on the single subflow without any MPTCP options.

If a subflow breaks during operation, e.g. if it is re-routed and MPTCP options are no longer permitted, then once this is detected (by the subflow-level receive buffer filling up, since there is no mapping available in order to DATA_ACK this data), the subflow SHOULD be treated as broken and closed with a RST, since no data can be delivered to the application layer, and no fallback signal can be reliably sent. This RST SHOULD include the MP_TCPRST option (Section 3.6) with a "Middlebox interference" reason code.

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.

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

Note that if checksum usage has not been negotiated, this fallback mechanism cannot be used unless there is some higher or lower layer signal to inform the MPTCP implementation that the payload has been tampered with.

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 adjustment has done to the data (notably, any changes to the subflow sequence numbering). Therefore, it is not possible to recover the subflow, and the affected subflow must be immediately closed with a RST, featuring an MP_FAIL option (Figure 16), which defines the data sequence number at the start of the segment (defined by the data sequence mapping) that had the checksum failure. Note that the MP_FAIL option requires the use of the full 64-bit sequence number, even if 32-bit sequence numbers are normally in use in the DSS signals on the path.

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

Figure 16: Fallback (MP_FAIL) Option

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

A special case is when there is a single subflow and it fails with a checksum error. If it is known that all unacknowledged data in flight is contiguous (which will usually be the case with a single subflow), an infinite mapping can be applied to the subflow without the need to close it first, and essentially turn off all further MPTCP signaling. In this case, if a receiver identifies a checksum failure when there is only one path, it will send back an MP_FAIL option on the subflow-level ACK, referring to the data-level sequence number of the start of the segment on which the checksum error was detected. The sender will receive this, and if all unacknowledged data in flight is contiguous, will signal an infinite mapping. This infinite mapping will be a DSS option (Section 3.3) on the first new packet, containing a data sequence mapping that acts retroactively, referring to the start of the subflow sequence number of the most recent segment that was known to be delivered intact (i.e. was successfully DATA_ACKed). 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. Whilst in theory paths may only be damaged in one direction, and the MP_FAIL signal affects only one direction of traffic, for implementation simplicity, the receiver of an MP_FAIL MUST also respond with an MP_FAIL in the reverse direction and entirely revert to a regular TCP session.

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

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

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

It should be emphasized that MPTCP is 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.

3.8. Error Handling

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

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

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

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

- Remove request for unknown address ID: silently ignore

3.9. Heuristics

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

3.9.1. 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 a host wishes to signal that a specific port should be used, and this facility is provided in the ADD_ADDR 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 [RFC2992]).

3.9.2. Delayed Subflow Start and Subflow Symmetry

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. Experimental deployments have shown that MPTCP can be applied in a range of scenarios so an implementation is likely to need to take into account factors including the type of traffic being sent and duration of session, and this information MAY be signalled by the application layer.

However, for standard TCP traffic, a suggested general-purpose heuristic that an implementation MAY choose to employ is as follows.

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

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

Note that this heuristic alone is probably insufficient. Traffic for many common applications, such as downloads, is highly asymmetric and the host that is multihomed may well be the client that will never fill its buffers, and thus never use MPTCP according to this heuristic. 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.

Another issue is that both communicating hosts may simultaneously try to set up a subflow between the same pair of addresses. This leads to an inefficient use of resources.

If the same ports are used on all subflows, as recommended above, then standard TCP simultaneous open logic should take care of this situation and only one subflow will be established between the address pairs. However, this relies on the same ports being used at both end hosts. If a host does not support TCP simultaneous open, it is RECOMMENDED that some element of randomization is applied to the time to wait before opening new subflows, so that only one subflow is created between a given address pair. If, however, hosts signal additional ports to use (for example, for leveraging ECMP on-path), this heuristic is not appropriate.

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

3.9.3. Failure Handling

Requirements for MPTCP's handling of unexpected signals have been given in Section 3.8. There are other failure cases, however, where a hosts can choose appropriate behavior.

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

Another failure could occur when the MP_JOIN handshake fails. Section 3.8 specifies that an incorrect handshake MUST lead to the subflow being closed with a RST. A host operating an active intrusion detection system may choose to start blocking MP_JOIN packets from the source host if multiple failed MP_JOIN attempts are seen. From the connection initiator's point of view, if an MP_JOIN fails, it SHOULD NOT attempt to connect to the same IP address and port during the lifetime of the connection, unless the other host refreshes the information with another ADD_ADDR option. Note that the ADD_ADDR option is informational only, and does not guarantee the other host will attempt a connection.

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

4. Semantic Issues

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

Sequence number: The (in-header) TCP sequence number is specific to the subflow. To allow the receiver to reorder application data, an additional data-level sequence space is used. In this data-level sequence space, the initial SYN and the final DATA_FIN occupy 1 octet of sequence space. This is to ensure these signals are acknowledged at the connection level. There is an explicit mapping of data sequence space to subflow sequence space, which is signaled through TCP options in data packets.

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

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

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

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

RST: The RST flag in the TCP header applies only to the subflow it is sent on, not to the whole connection. The MP_FASTCLOSE option provides the fast close functionality of a RST at the MPTCP connection level.

Address List: Address list management (i.e., knowledge of the local and remote hosts' lists of available IP addresses) is handled on a per-connection basis (as opposed to per subflow, per host, or per pair of communicating hosts). This permits the application of per-connection local policy. Adding an address to one connection (either explicitly through an Add Address message, or implicitly through a Join) has no implication for other connections between the same pair of hosts.

5-tuple: The 5-tuple (protocol, local address, local port, remote address, remote port) presented by kernel APIs to the application layer in a non-multipath-aware application is that of the first subflow, even if the subflow has since been closed and removed from the connection. This decision, and other related API issues, are discussed in more detail in [RFC6897].

5. Security Considerations

As identified in [RFC6181], the addition of multipath capability to TCP will bring with it a number of new classes of threat. In order to prevent these, [RFC6182] presents a set of requirements for a security solution for MPTCP. The fundamental goal is for the security of MPTCP to be "no worse" than regular TCP today, and the key security requirements are:

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

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

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

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

The security of the MPTCP connection hangs on the use of keys that are shared once at the start of the first subflow, and are never sent again over the network (unless used in the fast close mechanism, Section 3.5). To ease demultiplexing while not giving away any cryptographic material, future subflows use a truncated cryptographic hash of this key as the connection identification "token". The keys are concatenated and used as keys for creating Hash-based Message Authentication Codes (HMACs) used on subflow setup, in order to verify that the parties in the handshake are the same as in the original connection setup. It also provides verification that the peer can receive traffic at this new address. Replay attacks would still be possible when only keys are used; therefore, the handshakes use single-use random numbers (nonces) at both ends -- this ensures the HMAC will never be the same on two handshakes. Guidance on generating random numbers suitable for use as keys is given in [RFC4086] and discussed in Section 3.1. The nonces are valid for the lifetime of the TCP connection attempt. HMAC is also used to secure the ADD_ADDR option, due to the threats identified in [RFC7430].

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. This negotiation would nevertheless be susceptible to a bid-down attack by an on-path active attacker who could modify the crypto capability bits in the response from the receiver to use a less secure crypto mechanism. The security mechanism presented in this document should therefore protect against all forms of flooding and hijacking attacks discussed in [RFC6181].

The version negotiation specified in Section 3.1, if differing MPTCP versions shared a common negotiation format, would allow an on-path attacker to apply a theoretical bid-down attack. Since the v1 and v0 protocols have a different handshake, such an attack would require the client to re-establish the connection using v0, and this being supported by the server. Note that an on-path attacker would have access to the raw data, negating any other TCP-level security mechanisms. Also a change from RFC6824 has removed the subflow identifier from the MP_PRIO option (Section 3.3.8), to remove the theoretical attack where a subflow could be placed in "backup" mode by an attacker.

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

As discussed in Section 3.4.1, a host may advertise its private addresses, but these might point to different hosts in the receiver's network. The MP_JOIN handshake (Section 3.2) will ensure that this does not succeed in setting up a subflow to the incorrect host. However, it could still create unwanted TCP handshake traffic. This feature of MPTCP could be a target for denial-of-service exploits, with malicious participants in MPTCP connections encouraging the recipient to target other hosts in the network. Therefore, implementations should consider heuristics (Section 3.9) at both the sender and receiver to reduce the impact of this.

To further protect against malicious ADD_ADDR messages sent by an off-path attacker, the ADD_ADDR includes an HMAC using the keys negotiated during the handshake. This effectively prevents an attacker from diverting an MPTCP connection through an off-path ADD_ADDR injection into the stream.

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

Whilst this specification defines a "medium" security solution, meeting the criteria specified at the start of this section and the threat analysis ([RFC6181]), since attacks only ever get worse, it is likely that a future version of MPTCP would need to be able to support stronger security. There are several ways the security of MPTCP could potentially be improved; some of these would be compatible with MPTCP as defined in this document, whilst others may not be. For now, the best approach is to get experience with the current approach, establish what might work, and check that the threat analysis is still accurate.

Possible ways of improving MPTCP security could include:

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

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

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

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

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

- available flags in MP_CAPABLE (Figure 4);

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

- the version field in MP_CAPABLE (Figure 4);

6. Interactions with Middleboxes

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

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

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

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

If a middlebox strips options but otherwise passes the packets unchanged, MPTCP will behave safely. If an MP_CAPABLE option is dropped on either the outgoing or the return path, the initiating host can fall back to regular TCP, as illustrated in Figure 17 and discussed in Section 3.1.

Subflow SYNs contain the MP_JOIN option. If this option is stripped on the outgoing path, the SYN will appear to be a regular SYN to Host B. Depending on whether there is a listening socket on the target port, Host B will reply either with SYN/ACK or RST (subflow connection fails). When Host A receives the SYN/ACK it sends a RST because the SYN/ACK does not contain the MP_JOIN option and its token. Either way, the subflow setup fails, but otherwise does not affect the MPTCP connection as a whole.

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

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

Figure 17: Connection Setup with Middleboxes that Strip Options from Packets

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

The case when options get stripped on data packets has been discussed in the Fallback section. If only some MPTCP options are stripped, behavior is not deterministic. If some data sequence mappings are lost, the connection can continue so long as mappings exist for the subflow-level data (e.g., if multiple maps have been sent that reinforce each other). If some subflow-level space is left unmapped, however, the subflow is treated as broken and is closed, through the process described in Section 3.7. MPTCP should survive with a loss of some Data ACKs, but performance will degrade as the fraction of stripped options increases. We do not expect such cases to appear in practice, though: most middleboxes will either strip all options or let them all through.

We end this section with a list of middlebox classes, their behavior, and the elements in the MPTCP design that allow operation through such middleboxes. Issues surrounding dropping packets with options or stripping options were discussed above, and are not included here:

- NATs [RFC3022] (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 [RFC1918], since they are authenticated, will only set up subflows to the correct hosts. Explicit address removal is undertaken by an Address ID to allow no knowledge of the source address.

- Performance Enhancing Proxies (PEPs) [RFC3135] might proactively ACK data to increase performance. MPTCP, however, relies on accurate congestion control signals from the end host, and non-MPTCP-aware PEPs will not be able to provide such signals. MPTCP will, therefore, fall back to single-path TCP, or close the problematic subflow (see Section 3.7).

- Traffic Normalizers [norm] 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.

- Firewalls [RFC2979] might perform initial sequence number randomization on TCP connections. MPTCP uses relative sequence numbers in data sequence mapping to cope with this. Like NATs, firewalls will not permit many incoming connections, so MPTCP supports address signaling (ADD_ADDR) so that a multiaddressed host can invite its peer behind the firewall/NAT to connect out to its additional interface.

- Intrusion Detection/Prevention Systems (IDS/IPS) observe packet streams for patterns and content that could threaten a network. MPTCP may require the instrumentation of additional paths, and an MPTCP-aware IDS/IPS would need to read MPTCP tokens to correlate data from mutliple subflows to maintain comparable visibility into all of the traffic between devices. Without such changes, an IDS would get an incomplete view of the traffic, increasing the risk of missing traffic of interest (false negatives), and increasing the chances of erroneously identifying a subflow as a risk due to only seeing partial data (false positives).

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

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

- 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.

- 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.

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

7. Acknowledgments

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

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

Kuehlewind, Sheng Jiang, Alissa Cooper, Ines Robles, Roman Danyliw, Adam Roach, Barry Leiba, Alexey Melnikov, Eric Vyncke, and Ben Kaduk.

8. IANA Considerations

This document obsoletes RFC6824 and as such IANA is requested to update the TCP option space registry to point to this document for Multipath TCP, as follows:

Kind	Length	Meaning	Reference
30	N	Multipath TCP (MPTCP)	This document

Table 1: TCP Option Kind Numbers

8.1. MPTCP Option Subtypes

The 4-bit MPTCP subtype sub-registry ("MPTCP Option Subtypes" under the "Transmission Control Protocol (TCP) Parameters" registry) was defined in RFC6824. Since RFC6824 was an Experimental not Standards Track RFC, and since no further entries have occurred beyond those pointing to RFC6824, IANA is requested to replace the existing registry with Table 2 and with the following explanatory note.

Note: This registry specifies the MPTCP Option Subtypes for MPTCP v1, which obsoletes the Experimental MPTCP v0. For the MPTCP v0 subtypes, please refer to RFC6824.

Value	Symbol	Name	Reference
0x0	MP_CAPABLE	Multipath Capable	This document, Section 3.1
0x1	MP_JOIN	Join Connection	This document, Section 3.2
0x2	DSS	Data Sequence Signal (Data ACK and data sequence mapping)	This document, Section 3.3
0x3	ADD_ADDR	Add Address	This document, Section 3.4.1
0x4	REMOVE_ADDR	Remove Address	This document, Section 3.4.2
0x5	MP_PRIO	Change Subflow Priority	This document, Section 3.3.8
0x6	MP_FAIL	Fallback	This document, Section 3.7
0x7	MP_FASTCLOSE	Fast Close	This document, Section 3.5
0x8	MP_TCPRST	Subflow Reset	This document, Section 3.6
0xf	MP_EXPERIMENTAL	Reserved for private experiments	

Table 2: MPTCP Option Subtypes

Values 0x9 through 0xe are currently unassigned. Option 0xf is reserved for use by private experiments. Its use may be formalized in a future specification. Future assignments in this registry are to be defined by Standards Action as defined by [RFC8126]. Assignments consist of the MPTCP subtype's symbolic name and its associated value, and a reference to its specification.

8.2. MPTCP Handshake Algorithms

The "MPTCP Handshake Algorithms" sub-registry under the "Transmission Control Protocol (TCP) Parameters" registry was defined in RFC6824. Since RFC6824 was an Experimental not Standards Track RFC, and since no further entries have occurred beyond those pointing to RFC6824, IANA is requested to replace the existing registry with Table 3 and with the following explanatory note.

Note: This registry specifies the MPTCP Handshake Algorithms for MPTCP v1, which obsoletes the Experimental MPTCP v0. For the MPTCP v0 subtypes, please refer to RFC6824.

Flag Bit	Meaning	Reference
A	Checksum required	This document, Section 3.1
B	Extensibility	This document, Section 3.1
C	Do not attempt to establish new subflows to the source address.	This document, Section 3.1
D-G	Unassigned	
H	HMAC-SHA256	This document, Section 3.2

Table 3: MPTCP Handshake Algorithms

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

Future assignments in this registry are also to be defined by Standards Action as defined by [RFC8126]. Assignments consist of the value of the flags, a symbolic name for the algorithm, and a reference to its specification.

8.3. MP_TCPRST Reason Codes

IANA is requested to create a further sub-registry, "MPTCP MP_TCPRST Reason Codes" under the "Transmission Control Protocol (TCP) Parameters" registry, based on the reason code in MP_TCPRST (Section 3.6) message. Initial values for this registry are given in Table 4; future assignments are to be defined by Specification Required as defined by [RFC8126]. Assignments consist of the value of the code, a short description of its meaning, and a reference to its specification. The maximum value is 0xff.

As guidance to the Designated Expert [RFC8126], assignments should not normally be refused unless codepoint space is becoming scarce, providing that there is a clear distinction from other, already-existing codes, and also providing there is sufficient guidance for implementors both sending and receiving these codes.

Code	Meaning	Reference
0x00	Unspecified TCP error	This document, Section 3.6
0x01	MPTCP specific error	This document, Section 3.6
0x02	Lack of resources	This document, Section 3.6
0x03	Administratively prohibited	This document, Section 3.6
0x04	Too much outstanding data	This document, Section 3.6
0x05	Unacceptable performance	This document, Section 3.6
0x06	Middlebox interference	This document, Section 3.6

Table 4: MPTCP MP_TCPRST Reason Codes

9. References

9.1. Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC5961] Ramaiah, A., Stewart, R., and M. Dalal, "Improving TCP's Robustness to Blind In-Window Attacks", RFC 5961, DOI 10.17487/RFC5961, August 2010, <https://www.rfc-editor.org/info/rfc5961>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

9.2. Informative References

[deployments]
Bonaventure, O. and S. Seo, "Multipath TCP Deployments", IETF Journal 2016, November 2016, <https://www.ietfjournal.org/multipath-tcp-deployments/>.

[howhard] Raiciu, C., Paasch, C., Barre, S., Ford, A., Honda, M., Duchene, F., Bonaventure, O., and M. Handley, "How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP", Usenix Symposium on Networked Systems Design and Implementation 2012, 2012, <https://www.usenix.org/conference/nsdi12/how-hard-can-it-be-designing-and-implementing-deployable-multipath-tcp>.

[norm] 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>.

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <https://www.rfc-editor.org/info/rfc1122>.

[RFC1918] Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G., and E. Lear, "Address Allocation for Private Internets", BCP 5, RFC 1918, DOI 10.17487/RFC1918, February 1996, <https://www.rfc-editor.org/info/rfc1918>.

[RFC2018] Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, "TCP Selective Acknowledgment Options", RFC 2018, DOI 10.17487/RFC2018, October 1996, <https://www.rfc-editor.org/info/rfc2018>.

[RFC2979] Freed, N., "Behavior of and Requirements for Internet Firewalls", RFC 2979, DOI 10.17487/RFC2979, October 2000, <https://www.rfc-editor.org/info/rfc2979>.

[RFC2992] Hopps, C., "Analysis of an Equal-Cost Multi-Path Algorithm", RFC 2992, DOI 10.17487/RFC2992, November 2000, <https://www.rfc-editor.org/info/rfc2992>.

[RFC3022] Srisuresh, P. and K. Egevang, "Traditional IP Network Address Translator (Traditional NAT)", RFC 3022, DOI 10.17487/RFC3022, January 2001, <https://www.rfc-editor.org/info/rfc3022>.

[RFC3135] Border, J., Kojo, M., Griner, J., Montenegro, G., and Z. Shelby, "Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations", RFC 3135, DOI 10.17487/RFC3135, June 2001, <https://www.rfc-editor.org/info/rfc3135>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

[RFC4987] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", RFC 4987, DOI 10.17487/RFC4987, August 2007, <https://www.rfc-editor.org/info/rfc4987>.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <https://www.rfc-editor.org/info/rfc5681>.

[RFC6181] Bagnulo, M., "Threat Analysis for TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6181, DOI 10.17487/RFC6181, March 2011, <https://www.rfc-editor.org/info/rfc6181>.

[RFC6182] Ford, A., Raiciu, C., Handley, M., Barre, S., and J. Iyengar, "Architectural Guidelines for Multipath TCP Development", RFC 6182, DOI 10.17487/RFC6182, March 2011, <https://www.rfc-editor.org/info/rfc6182>.

[RFC6356] Raiciu, C., Handley, M., and D. Wischik, "Coupled Congestion Control for Multipath Transport Protocols", RFC 6356, DOI 10.17487/RFC6356, October 2011, <https://www.rfc-editor.org/info/rfc6356>.

[RFC6528] Gont, F. and S. Bellovin, "Defending against Sequence Number Attacks", RFC 6528, DOI 10.17487/RFC6528, February 2012, <https://www.rfc-editor.org/info/rfc6528>.

[RFC6897] Scharf, M. and A. Ford, "Multipath TCP (MPTCP) Application Interface Considerations", RFC 6897, DOI 10.17487/RFC6897, March 2013, <https://www.rfc-editor.org/info/rfc6897>.

[RFC7323] Borman, D., Braden, B., Jacobson, V., and R. Scheffenegger, Ed., "TCP Extensions for High Performance", RFC 7323, DOI 10.17487/RFC7323, September 2014, <https://www.rfc-editor.org/info/rfc7323>.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <https://www.rfc-editor.org/info/rfc7413>.

[RFC7430] Bagnulo, M., Paasch, C., Gont, F., Bonaventure, O., and C. Raiciu, "Analysis of Residual Threats and Possible Fixes for Multipath TCP (MPTCP)", RFC 7430, DOI 10.17487/RFC7430, July 2015, <https://www.rfc-editor.org/info/rfc7430>.

[RFC8041] Bonaventure, O., Paasch, C., and G. Detal, "Use Cases and Operational Experience with Multipath TCP", RFC 8041, DOI 10.17487/RFC8041, January 2017, <https://www.rfc-editor.org/info/rfc8041>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

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

Appendix A. Notes on Use of TCP Options

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

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

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

Note that due to the use of a 64-bit data-level sequence space, it is feasible that MPTCP will not require the timestamp option for protection against wrapped sequence numbers (PAWS [RFC7323]), since the data-level sequence space has far less chance of wrapping. Confirmation of the validity of this optimisation is for further study.

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

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

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

Pure ACKs in TCP typically contain only timestamps (10 bytes). Here, Multipath TCP typically needs to encode only the DATA_ACK (maximum of 12 bytes). Occasionally, ACKs will contain SACK information. Depending on the number of lost packets, SACK may utilize the entire option space. If a DATA_ACK had to be included, then it is probably necessary to reduce the number of SACK blocks to accommodate the DATA_ACK. However, the presence of the DATA_ACK is unlikely to be necessary in a case where SACK is in use, since until at least some of the SACK blocks have been retransmitted, the cumulative data-level ACK will not be moving forward (or if it does, due to retransmissions on another path, then that path can also be used to transmit the new DATA_ACK).

The ADD_ADDR option can be between 16 and 30 bytes, depending on whether IPv4 or IPv6 is used, and whether or not the port number is present. It is unlikely that such signaling would fit in a data packet (although if there is space, it is fine to include it). It is recommended to use duplicate ACKs with no other payload or options in order to transmit these rare signals. Note this is the reason for mandating that duplicate ACKs with MPTCP options are not taken as a signal of congestion.

Appendix B. TCP Fast Open and MPTCP

TCP Fast Open (TFO) is an experimental TCP extension, described in [RFC7413], which has been introduced to allow sending data one RTT earlier than with regular TCP. This is considered a valuable gain as very short connections are very common, especially for HTTP request/response schemes. It achieves this by sending the SYN-segment together with the application's data and allowing the listener to reply immediately with data after the SYN/ACK. [RFC7413] secures this mechanism, by using a new TCP option that includes a cookie which is negotiated in a preceding connection.

When using TCP Fast Open in conjunction with MPTCP, there are two key points to take into account, detailed hereafter.

B.1. TFO cookie request with MPTCP

When a TFO initiator first connects to a listener, it cannot immediately include data in the SYN for security reasons [RFC7413]. Instead, it requests a cookie that will be used in subsequent connections. This is done with the TCP cookie request/response options, of respectively 2 bytes and 6-18 bytes (depending on the chosen cookie length).

TFO and MPTCP can be combined provided that the total length of all the options does not exceed the maximum 40 bytes possible in TCP:

- In the SYN: MPTCP uses a 4-bytes long MP_CAPABLE option. The MPTCP and TFO options sum up to 6 bytes. With typical TCP-options using up to 19 bytes in the SYN (24 bytes if options are padded at a word boundary), there is enough space to combine the MP_CAPABLE with the TFO Cookie Request.

- In the SYN+ACK: MPTCP uses a 12-bytes long MP_CAPABLE option, but now TFO can be as long as 18 bytes. Since the maximum option length may be exceeded, it is up to the listener to solve this by using a shorter cookie. As an example, if we consider that 19 bytes are used for classical TCP options, the maximum possible cookie length would be of 7 bytes. Note that the same limitation applies to subsequent connections, for the SYN packet (because the initiator then echoes back the cookie to the listener). Finally, if the security impact of reducing the cookie size is not deemed acceptable, the listener can reduce the amount of other TCP-options by omitting the TCP timestamps (as outlined in Appendix A).

B.2. Data sequence mapping under TFO

MPTCP uses, in the TCP establishment phase, a key exchange that is used to generate the Initial Data Sequence Numbers (IDSNs). In particular, the SYN with MP_CAPABLE occupies the first octet of the data sequence space. With TFO, one way to handle the data sent together with the SYN would be to consider an implicit DSS mapping that covers that SYN segment (since there is not enough space in the SYN to include a DSS option). The problem with that approach is that if a middlebox modifies the TFO data, this will not be noticed by MPTCP because of the absence of a DSS-checksum. For example, a TCP (but not MPTCP)-aware middlebox could insert bytes at the beginning of the stream and adapt the TCP checksum and sequence numbers accordingly. With an implicit mapping, this would give to initiator and listener a different view on the DSS-mapping, with no way to detect this inconsistency as the DSS checksum is not present.

To solve this, the TFO data must not be considered part of the Data Sequence Number space: the SYN with MP_CAPABLE still occupies the first octet of data sequence space, but then the first non-TFO data byte occupies the second octet. This guarantees that, if the use of DSS-checksum is negotiated, all data in the data sequence number space is checksummed. We also note that this does not entail a loss of functionality, because TFO-data is always only sent on the initial subflow before any attempt to create additional subflows.

B.3. Connection establishment examples

The following shows a few examples of possible TFO+MPTCP establishment scenarios.

Before an initiator can send data together with the SYN, it must request a cookie to the listener, as shown in Figure 18. This is done by simply combining the TFO and MPTCP options.

```
initiator                                                    listener
    |                                                              |
    |    S Seq=0(Length=0) <MP_CAPABLE>, <TFO cookie request>      |
    | -----------------------------------------------------------> |
    |                                                              |
    |    S. 0(0) ack 1 <MP_CAPABLE>, <TFO cookie>                  |
    | <----------------------------------------------------------- |
    |                                                              |
    |    .  0(0) ack 1 <MP_CAPABLE>                                |
    | -----------------------------------------------------------> |
    |                                                              |
```

Figure 18: Cookie request - sequence number and length are annotated as Seq(Length) and used hereafter in the figures.

Once this is done, the received cookie can be used for TFO, as shown in Figure 19. In this example, the initiator first sends 20 bytes in the SYN. The listener immediately replies with 100 bytes following the SYN-ACK upon which the initiator replies with 20 more bytes. Note that the last segment in the figure has a TCP sequence number of 21, while the DSS subflow sequence number is 1 (because the TFO data is not part of the data sequence number space, as explained in Section Appendix B.2.

```
initiator                                                       listener
    |                                                              |
    |    S  0(20) <MP_CAPABLE>, <TFO cookie>                       |
    | -----------------------------------------------------------> |
    |                                                              |
    |    S. 0(0) ack 21 <MP_CAPABLE>                               |
    | <----------------------------------------------------------- |
    |                                                              |
    |    .  1(100) ack 21 <DSS ack=1 seq=1 ssn=1 dlen=100>         |
    | <----------------------------------------------------------- |
    |                                                              |
    |    .  21(0) ack 1 <MP_CAPABLE>                               |
    | -----------------------------------------------------------> |
    |                                                              |
    |    .  21(20) ack 101 <DSS ack=101 seq=1 ssn=1 dlen=20>       |
    | -----------------------------------------------------------> |
    |                                                              |
```

Figure 19: The listener supports TFO

In Figure 20, the listener does not support TFO. The initiator detects that no state is created in the listener (as no data is acked), and now sends the MP_CAPABLE in the third ack, in order for the listener to build its MPTCP context at then end of the establishment. Now, the tfo data, retransmitted, becomes part of the data sequence mapping because it is effectively sent (in fact re-sent) after the establishment.

```
initiator                                                       listener
    |                                                              |
    |    S  0(20) <MP_CAPABLE>, <TFO cookie>                       |
    | -----------------------------------------------------------> |
    |                                                              |
    |    S. 0(0) ack 1 <MP_CAPABLE>                                |
    | <----------------------------------------------------------- |
    |                                                              |
    |    .  1(0) ack 1 <MP_CAPABLE>                                |
    | -----------------------------------------------------------> |
    |                                                              |
    |    .  1(20) ack 1 <DSS ack=1 seq=1 ssn=1 dlen=20>            |
    | -----------------------------------------------------------> |
    |                                                              |
    |    .  0(0) ack 21 <DSS ack=21 seq=1 ssn=1 dlen=0>            |
    | <----------------------------------------------------------- |
    |                                                              |
```

Figure 20: The listener does not support TFO

It is also possible that the listener acknowledges only part of the TFO data, as illustrated in Figure 21. The initiator will simply retransmit the missing data together with a DSS-mapping.

```
initiator                                                     listener
    |                                                              |
    |  S  0(1000) <MP_CAPABLE>, <TFO cookie>                       |
    | ------------------------------------------------------------> |
    |                                                              |
    |  S. 0(0) ack 501 <MP_CAPABLE>                                |
    | <------------------------------------------------------------ |
    |                                                              |
    |    .  501(0) ack 1 <MP_CAPABLE>                              |
    | ------------------------------------------------------------> |
    |                                                              |
    |   .  501(500) ack 1 <DSS ack=1 seq=1 ssn=1 dlen=500>         |
    | ------------------------------------------------------------> |
    |                                                              |
```

Figure 21: Partial data acknowledgement

Appendix C. Control Blocks

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

RFC 793 [RFC0793] specifies several state variables. Whenever possible, we reuse the same terminology as RFC 793 to describe the state variables that are maintained by MPTCP.

C.1. MPTCP Control Block

The MPTCP control block contains the following variable per connection.

C.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, and is 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 A 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.

C.1.2. Sending Side

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

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

SND.WND (32 bits with RFC 7323, 16 bits otherwise): This is the sending window. MPTCP maintains the sending window at the MPTCP connection level and the same window is shared by all subflows. All subflows use the MPTCP connection level SND.WND to compute the SEQ.WND value that is sent in each transmitted segment.

C.1.3. Receiving Side

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

RCV.WND (32 bits with RFC 7323, 16 bits otherwise): This is the connection-level receive window, which is the maximum of the RCV.WND on all the subflows.

C.2. TCP Control Blocks

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

C.2.1. Sending Side

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

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

C.2.2. Receiving Side

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

RCV.WND (32 bits with RFC 7323, 16 bits otherwise): This is the subflow-level receive window that is updated with the window field from the segments received on this subflow.

Appendix D. Finite State Machine

The diagram in Figure 22 shows the Finite State Machine for connection-level closure. This illustrates how the DATA_FIN connection-level signal (indicated in the diagram as the DFIN flag on a DATA_ACK) interacts with subflow-level FINs, and permits "break-before-make" handover between subflows.

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

Figure 22: Finite State Machine for Connection Closure

Appendix E. Changes from RFC6824

This section lists the key technical changes between RFC6824, specifying MPTCP v0, and this document, which obsoletes RFC6824 and specifies MPTCP v1. Note that this specification is not backwards compatible with RFC6824.

- o The document incorporates lessons learnt from the various implementations, deployments and experiments gathered in the documents "Use Cases and Operational Experience with Multipath TCP" [RFC8041] and the IETF Journal article "Multipath TCP Deployments" [deployments].

- o Connection initiation, through the exchange of the MP_CAPABLE MPTCP option, is different from RFC6824. The SYN no longer includes the initiator's key, allowing the MP_CAPABLE option on the SYN to be shorter in length, and to avoid duplicating the sending of keying material.

- This also ensures reliable delivery of the key on the MP_CAPABLE option by allowing its transmission to be combined with data and thus using TCP's in-built reliability mechanism. If the initiator does not immediately have data to send, the MP_CAPABLE option with the keys will be repeated on the first data packet. If the other end is first to send, then the presence of the DSS option implicitly confirms the receipt of the MP_CAPABLE.

- In the Flags field of MP_CAPABLE, C is now assigned to mean that the sender of this option will not accept additional MPTCP subflows to the source address and port. This is an efficiency improvement, for example where the sender is behind a strict NAT.

- In the Flags field of MP_CAPABLE, H now indicates the use of HMAC-SHA256 (rather than HMAC-SHA1).

- Connection initiation also defines the procedure for version negotiation, for implementations that support both v0 (RFC6824) and v1 (this document).

- The HMAC-SHA256 (rather than HMAC-SHA1) algorithm is used, as the algorithm provides better security. It is used to generate the token in the MP_JOIN and ADD_ADDR messages, and to set the initial data sequence number.

- A new subflow-level option exists to signal reasons for sending a RST on a subflow (MP_TCPRST Section 3.6), which can help an implementation decide whether to attempt later re-connection.

- The MP_PRIO option (Section 3.3.8), which is used to signal a change of priority for a subflow, no longer includes the AddrID field. Its purpose was to allow the changed priority to be applied on a subflow other than the one it was sent on. However, it has been realised that this could be used by a man-in-the-middle to divert all traffic on to its own path, and MP_PRIO does not include a token or other security mechanism.

- The ADD_ADDR option (Section 3.4.1), which is used to inform the other host about another potential address, is different in several ways. It now includes an HMAC of the added address, for enhanced security. In addition, reliability for the ADD_ADDR option has been added: the IPVer field is replaced with a flag field, and one flag is assigned (E) which is used as an 'Echo' so a host can indicate that it has received the option.

- An additional way of performing a Fast Close is described, by sending a MP_FASTCLOSE option on a RST on all subflows. This allows the host to tear down the subflows and the connection immediately.

- In the IANA registry a new MPTCP subtype option, MP_EXPERIMENTAL, is reserved for private experiments. However, the document doesn't define how to use the subtype option.

- A new Appendix discusses the usage of both the MPTCP and TCP Fast Open on the same packet (Appendix B).

Authors' Addresses

Alan Ford
Pexip

EMail: alan.ford@gmail.com

Costin Raiciu
University Politehnica of Bucharest
Splaiul Independentei 313
Bucharest
Romania

EMail: costin.raiciu@cs.pub.ro

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

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

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

EMail: olivier.bonaventure@uclouvain.be

Christoph Paasch
Apple, Inc.
Cupertino
US

EMail: cpaasch@apple.com

Internet Area Working Group V. Olteanu
Internet-Draft D. Niculescu
Intended status: Experimental University Politehnica of Bucharest
Expires: May 3, 2018 October 30, 2017

SOCKS Protocol Version 6
draft-olteanu-intarea-socks-6-01

Abstract

The SOCKS protocol is used primarily to proxy TCP connections to arbitrary destinations via the use of a proxy server. Under the latest version of the protocol (version 5), it takes 2 RTTs (or 3, if authentication is used) before data can flow between the client and the server.

This memo proposes SOCKS version 6, which reduces the number of RTTs used, takes full advantage of TCP Fast Open, and adds support for 0-RTT authentication.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Versions 4 and 5 [RFC1928] of the SOCKS protocol were developed two decades ago and are in widespread use for circuit level gateways or as circumvention tools, and enjoy wide support and usage from various software, such as web browsers, SSH clients, and proxifiers. However, their design needs an update in order to take advantage of the new features of transport protocols, such as TCP Fast Open [RFC7413], or to better assist newer transport protocols, such as MPTCP [RFC6824].

One of the main issues faced by SOCKS version 5 is that, when taking into account the TCP handshake, method negotiation, authentication, connection request and grant, it may take up to 5 RTTs for a data exchange to take place at the application layer. This is especially costly in networks with a large delay at the access layer, such as 3G, 4G, or satelite.

The desire to reduce the number of RTTs manifests itself in the design of newer security protocols. TLS version 1.3 [I-D.ietf-tls-tls13] defines a zero round trip (0-RTT) handshake mode for connections if the client and server had previously communicated.

TCP Fast Open [RFC7413] is a TCP option that allows TCP to send data in the SYN and receive a response in the first ACK, and aims at obtaining a data response in one RTT. The SOCKS protocol needs to concern itself with at least two TFO deployment scenarios: First, when TFO is available end-to-end (at the client, at the proxy, and at the server); second, when TFO is active between the client and the proxy, but not at the server.

This document describes the SOCKS protocol version 6. The key improvements over SOCKS version 5 are:

- The client sends as much information upfront as possible, and does not wait for the authentication process to conclude before requesting the creation of a socket.

- The connection request also mimics the semantics of TCP Fast Open [RFC7413]. As part of the connection request, the client can supply the payload for the initial SYN that is sent out to the server.

- The protocol can be extended via options without breaking backward-compatibility.

- The protocol can leverage the aforementioned options to support 0-RTT authentication schemes.

1.1. Revision log

draft-01

- Added this section.

- Support for idempotent commands.

- Removed version numbers from operation replies.

- Request port number for SOCKS over TLS. Deprecate encryption/ encapsulation within SOCKS.

   o Added Version Mismatch Replies.

   o Renamed the AUTH command to NOOP.

   o Shifted some fields to make requests and operation replies easier
      to parse.

2. 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 [RFC2119].

3. Mode of operation

```
    CLIENT                                                        PROXY

            +------------------------+
            | Authentication methods | Request
    --------> Command code           +------------------------------->
            | TFO                    |
            | Address                |
            | Port                   |
            | Options                |
            | Initial data           |
            +------------------------+

                                        +-----------------------+
                   Authentication reply | Type                  |
     <----------------------------------+ Method                <-----
                                        | Options               |
                                        +-----------------------+

     <-------------------(Authentication protocol)------------------>

                          +-----------------------+
       Operation reply    | Reply code            |
     <--------------------+ Bind address          <------------------
                          | Bind port             |
                          | Options               |
                          | Initial data offset   |
                          +-----------------------+
```

Figure 1: The SOCKS version 6 protocol message exchange

When a TCP-based client wishes to establish a connection to a server, it must open a TCP connection to the appropriate SOCKS port on the SOCKS proxy. The client then enters a negotiation phase, by sending the request in figure Figure 1, that contains, in addition to fields present in SOCKS 5 [RFC1928], fields that facilitate low RTT usage and faster authentication negotiation.

Next, the server sends an authentication reply. If the request did not contain the necessary authentication information, the proxy indicates an authentication method that must proceed. This may trigger a longer authentication sequence that could include tokens for ulterior faster authentications. The part labeled "Authentication protocol" is specific to the authentication method employed and is not expected to be employed for every connection between a client and its proxy server. The authentication protocol typically takes up 1 RTT or more.

If the authentication is successful, an operation reply is generated by the proxy. It indicates whether the proxy was successful in creating the requested socket or not.

In the fast case, when authentication is properly set up, the proxy attempts to create the socket immediately after the receipt of the request, thus achieving an operational conection in one RTT (provided TFO functionality is available at the client, proxy, and server).

4. Connection Requests

The client starts by sending a request to the proxy.

```
+---------------+-----------+----------+
|    Version    | Number of | Methods  |
| Major | Minor |  Methods  |          |
+-------+-------+-----------+----------+
|   1   |   1   |     1     | Variable |
+-------+-------+-----------+----------+
+---------+-----+---------+------+----------+
| Command | TFO | Address | Port | Address  |
|  Code   |     |  Type   |      |          |
+---------+-----+---------+------+----------+
|    1    |  1  |    1    |  2   | Variable |
+---------+-----+---------+------+----------+
+-----------+----------+--------------+--------------+
| Number of | Options  | Initial Data | Initial Data |
|  Options  |          |     Size     |              |
+-----------+----------+--------------+--------------+
|     1     | Variable |      2       |   Variable   |
+-----------+----------+--------------+--------------+
```

Figure 2: SOCKS 6 Request

- Version: The major byte MUST be set to 0x06, and the minor byte MUST be set to 0x00.

- Number of Methods: The number of supported authentication methods that the client wishes to advertise.

- Methods: One byte per advertised method. Method numbers are assigned by IANA.

- Command Code:

  * 0x00 NOOP: authenticate the client and do nothing.

  * 0x01 CONNECT: requests the establishment of a TCP connection.

  * 0x02 BIND: requests the establishment of a TCP port binding.

  * 0x03 UDP ASSOCIATE: requests a UDP port association.

- TFO:

  * 0x00 indicates that the proxy MUST NOT attempt to use TFO in case of a CONNECT command, or accept TFO in case of a BIND command. In case of an AUTH or UDP ASSOCIATE command, this field MUST be set to 0x00.

      * 0x01 indicates that the proxy SHOULD attempt to use TFO in case of a CONNECT command, or accept TFO in case of a BIND command.

- Address Type:

  * 0x01: IPv4

  * 0x03: Domain Name

  * 0x04: IPv6

- Address: this field's format depends on the address type:

  * IPv4: a 4-byte IPv4 address

  * Domain Name: one byte that contains the length of the FQDN, followed by the FQDN itself. The string is not NUL-terminated.

  * IPv6: a 16-byte IPv6 address

- Port: the port in network byte order.

- Number of Options: the number of SOCKS options that appear in the Options field.

- Options: see section Section 8.

- Initial Data Size: A two-byte number in network byte order. In case of AUTH, BIND or UDP ASSOCIATE, this field MUST be set to 0. In case of CONNECT, this is the number of bytes of initial data that are supplied in the following field.

- Initial Data: The first octets of the data stream.

Clients MUST support the "No authentication required" method. Clients MAY omit advertising the "No authentication required" option.

Clients SHOULD NOT issue AUTH commands unless they advertise authentication methods with support for 0-RTT authentication.

The server MAY truncate the initial data to an arbitrary size and disregard the rest. This is will be communicated later to the client, should the authentication process be successful (see section Section 7). As such, server implementations do not have to buffer the initial data while waiting for the (potentially malicious) client to authenticate.

5. Version Mismatch Replies

Upon receipt of a request starting with a version number other than 6.0, the proxy sends the following response:

Version	
Major	Minor
1	1

Figure 3: SOCKS 6 Version Mismatch Reply

- o Version: The major byte MUST be set to 0x06, and the minor byte MUST be set to 0x00.

A client MUST close the connection after receiving such a reply.

6. Authentication Replies

Upon receipt of a valid request, the proxy sends an Authentication Reply:

Version		Type	Method	Number of Options	Options
Major	Minor				
1	1	1	1	1	Variable

Figure 4: SOCKS 6 Authentication Reply

- o Version: The major byte MUST be set to 0x06, and the minor byte MUST be set to 0x00.
- o Type:
  - * 0x00: authentication successful.
  - * 0x01: further authentication needed.
- o Method: The chosen authentication method.
- o Number of Options: the number of SOCKS options that appear in the Options field.

o Options: see section Section 8.

Multihomed clients SHOULD cache the chosen method on a per-interface basis and SHOULD NOT include authentication options related to any other methods in further requests originating from the same interface.

If the server signals that further authentication is needed and selects "No Acceptable Methods", the client MUST close the connection.

The client and proxy begin a method-specific negotiation. During such negotiations, the proxy MAY supply information that allows the client to authenticate a future request using an authentication option. Descriptions of such negotiations are beyond the scope of this memo.

If the cliend issued an AUTH command, the client MUST close the connection after the negociation is complete.

7. Operation Replies

After the authentication negotiations are complete, the server sends an Operation Reply:

```
+-------+---------+------+----------+--------------+
| Reply | Address | Bind |   Bind   | Initial Data |
| Code  |  Type   | Port | Address  |    Offset    |
+-------+---------+------+----------+--------------+
|   1   |    1    |  2   | Variable |      2       |
+-------+---------+------+----------+--------------+
+-----------+----------+
| Number of | Options  |
|  Options  |          |
+-----------+----------+
|     1     | Variable |
+-----------+----------+
```

Figure 5: SOCKS 6 Operation Reply

o Reply Code:

   * 0x00: Succes

   * 0x01: General SOCKS server failure

   * 0x02: Connection not allowed by ruleset

* 0x03: Network unreachable

* 0x04: Host unreachable

* 0x05: Connection refused

* 0x06: TTL expired

* 0x07: Command not supported

* 0x08: Address type not supported

o Address Type:

* 0x01: IPv4

* 0x03: Domain Name

* 0x04: IPv6

o Bind Address: the proxy bound address in the following format:

* IPv4: a 4-byte IPv4 address

* Domain Name: one byte that contains the length of the FQDN, followed by the FQDN itself. The string is not NUL-terminated.

* IPv6: a 16-byte IPv6 address

o Bind Port: the proxy bound port in network byte order.

o Number of Options: the number of SOCKS options that appear in the Options field.

o Options: see section Section 8.

o Initial Data Offset: A two-byte number in network byte order. In case of BIND or UDP ASSOCIATE, this field MUST be set to 0. In case of CONNECT, it represents the offset in the plain data stream from which the client is expected to continue sending data.

If the proxy returns a reply code other than "Success", the client MUST close the connection.

7.1. Handling CONNECT

In case the client has issued a CONNECT request, data can now pass. The client MUST resume the data stream at the offset indicated by the Initial Data Offset field.

7.2. Handling BIND

In case the client has issued a BIND request, it must wait for a second Operation reply from the proxy, which signifies that a host has connected to the bound port. The Bind Address and Bind Port fields contain the address and port of the connecting host. Afterwards, application data may pass.

7.3. Handling UDP ASSOCIATE

The relay of UDP packets is handled exactly as in SOCKS 5 [RFC1928].

8. SOCKS Options

SOCKS options have the following format:

```
+---------------+-------------+
| Kind | Length | Option Data |
+------+--------+-------------+
|  1   |   1    |   Variable  |
+------+--------+-------------+
```

Figure 6: SOCKS 6 Option

- o Kind: MUST be allocated by IANA. (See section Section 10.)
- o Length: The length of the option.
- o Option Data: The contents are specific to each option kind.

8.1. Authentication options

Authentication options carry method-specific authentication data. They can be part of SOCKS Requests and Authentication Replies.

Authentication options have the following format:

```
+---------------+--------+--------+--------------------+
| Kind | Length | Method | Authentication Data |
+------+--------+--------+--------------------+
|  1   |   1    |   1    |      Variable      |
+------+--------+--------+--------------------+
```

Figure 7: Authentication Option

- Kind: MUST be allocated by IANA. (See section Section 10.)
- Length: The length of the option.
- Method: The number of the authentication method. These numbers are assigned by IANA.
- Authentication Data: The contents are specific to each method.

All proxy implementations MUST support authentication method options. Clients MAY omit advertising authentication methods for which they have included at least an authentication option.

8.2. Idempotence options

To protect against duplicate SOCKS Requests, authenticated clients can request, and then spend, idempotence tokens. A token can only be spent on a single SOCKS request.

Tokens are 4-byte unsigned integers in a modular 4-byte space. Therefore, if x and y are tokens, x is smaller than y if (y - x) < 2^31 in unsigned 32-bit arithmetic.

Proxies grant contiguous ranges of tokens called token windows. Token windows are defined by their base (the first token in the range) and size. Windows can be shifted (i. e. have their base increased, while retaining their size) unilaterally by the proxy.

Requesting and spending tokens is done via Idempotence options:

```
+---------------+------+------------+
| Kind | Length | Type | Option Data |
+------+--------+------+------------+
|  1   |   1    |  1   |  Variable  |
+------+--------+------+------------+
```

Figure 8: Idempotence Option

o Kind: MUST be allocated by IANA. (See section Section 10.)

o Length: The length of the option.

o Type:

  * 0x00: Token Request

  * 0x01: Token Window Advertisement

  * 0x02: Token Expenditure

  * 0x03: Token Expenditure Reply

o Option Data: The contents are specific to each type.

8.2.1. Requesting a fresh token window

A client can obtain a fresh window of tokens by sending a Token Request option as part of a SOCKS Request:

```
+---------------+------+-------------+
| Kind | Length | Type | Window Size |
+------+--------+------+-------------+
|  1   |   1    |  1   |      4      |
+------+--------+------+-------------+
```

Figure 9: Token Request

o Kind: MUST be allocated by IANA. (See section Section 10.)

o Length: 7

o Type: 0x00 (Token Request)

o Window Size: The requested window size.

The proxy then includes a Token Window Advertisement option in the corresponding Operation Reply:

```
+------+--------+------+-------------+-------------+
| Kind | Length | Type | Window Base | Window Size |
+------+--------+------+-------------+-------------+
|  1   |   1    |  1   |      4      |      4      |
+------+--------+------+-------------+-------------+
```

Figure 10: Token Window Advertisement

- Kind: MUST be allocated by IANA. (See section Section 10.)
- Length: 11
- Type: 0x01 (Token Grant)
- Window Base: The first token in the window.
- Window Size: The window size. This value SHOULD be lower or equal to the requested window size.

8.2.2. Spending a token

The client can attempt to spend a token by including a Token Expenditure option in its SOCKS request:

```
+------+--------+------+-------+
| Kind | Length | Type | Token |
+------+--------+------+-------+
|  1   |   1    |  1   |   4   |
+------+--------+------+-------+
```

Figure 11: Token Expenditure

- Kind: MUST be allocated by IANA. (See section Section 10.)
- Length: 7
- Type: 0x02 (Token Expenditure)
- Token: The token being spent.

Clients SHOULD prioritize spending the smaller tokens.

The server responds by sending a Token Expenditure Reply option as part of the Operation Reply:

```
+---------------+------+--------------+
| Kind | Length | Type | Response Code |
+------+--------+------+---------------+
|  1   |   1    |  1   |       1       |
+------+--------+------+---------------+
```

Figure 12: Token Expenditure Response

- Kind: MUST be allocated by IANA. (See section Section 10.)
- Length: 4
- Type: 0x03 (Token Expenditure Response)
- Response Code:
  * 0x00: Success: The token was spent successfully.
  * 0x01: No Window: The proxy does not have a token window associated with the client.
  * 0x02: Out of Window: The token is not within the window.
  * 0x03: Duplicate: The token has already been spent.

If eligible, the token is spent as soon as the client authenticates. If the token is not eligible for spending, the proxy MUST NOT attempt to honor the client’s SOCKS Request; further, it MUST indicate a General SOCKS server failure in the Operation Reply.

Proxy implementations SHOULD also send a Token Window Advertisement if:

- the token is out of window, or
- by the proxy’s internal logic, successfully spending the token caused the window to shift.

Proxy implementations SHOULD NOT shift the window’s base beyond the highest unspent token.

Proxy implementations MAY include a Token Window Advertisement in any Operation Reply.

8.2.3. Handling Token Window Advertisements

Even though the proxy increases the window's base monotonically, there is no mechanism whereby a SOCKS client can receive the Token Window Advertisements in order. As such, clients SHOULD disregard unsollicited Token Window Advertisements with a Window Base less than the previously known value.

9. Security Considerations

9.1. Large requests

Given the format of the request message, a malicious client could craft a request that is in excess of 100 KB and proxies could be prone to DDoS attacks.

To mitigate such attacks, proxy implementations SHOULD be able to incrementally parse the requests. Proxies MAY close the connection to the client if:

- the request is not fully received after a certain timeout, or
- the number of options exceeds an imposed hard cap, or
- the total size of the options exceeds an imposed hard cap, or
- the size of the initial data excedes a hard cap.

Further, the server MAY choose not to buffer any initial data beyond what would be expected to fit in a TFO SYN's payload.

9.2. Replay attacks

In TLS 1.3, early data (which is likely to contain a full SOCKS request) is prone to replay attacks.

While Token Expenditure options can be used to mitigate replay attacks, the initial Token Request is still vulnerable. As such, client implementations SHOULD NOT make use of TLS early data when sending a Token Request.

10. IANA Considerations

This document requests that IANA allocate option codes for SOCKS 6 options. Further, this document requests option codes for authentication and idempotence options.

This document also requests that IANA allocate a port for SOCKS over TLS.

11. Acknowledgements

The protocol described in this draft builds upon and is a direct continuation of SOCKS 5 [RFC1928].

12. References

12.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

12.2. Informative References

[I-D.ietf-tls-tls13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-21 (work in progress), July 2017.

[RFC1928] Leech, M., Ganis, M., Lee, Y., Kuris, R., Koblas, D., and L. Jones, "SOCKS Protocol Version 5", RFC 1928, DOI 10.17487/RFC1928, March 1996, <https://www.rfc-editor.org/info/rfc1928>.

[RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <https://www.rfc-editor.org/info/rfc6824>.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <https://www.rfc-editor.org/info/rfc7413>.

Authors' Addresses

Vladimir Olteanu
University Politehnica of Bucharest

Email: vladimir.olteanu@cs.pub.ro

Dragos Niculescu
University Politehnica of Bucharest

Email: dragos.niculescu@cs.pub.ro

MPTCP Working Group   Samar Shailendra
Internet-Draft   Hemant Kumar Rath
Intended status: Informational   Arpan Pal
Expires: February 6, 2020   TCS R&I
                                                          Abhijit Mondal
                                                          IIT Kharagpur
                                                          August 5, 2019

Extended Socket APIs to control subflow priority in Multipath TCP
draft-samar-mptcp-socketapi-03.txt

Status of this Memo

Copyright Notice

Abstract

This document provides the extended Socket APIs to control subflow priority for Multipath TCP. It also describes an additional data structure for MPTCP to make the subflow priority persistent across subflow disconnection.

Table of Contents

1. Introduction

Multipath TCP (MPTCP) [RFC6824, RFC6182] has been designed as a successor to TCP [RFC0793] with complete backward compatibility i.e. it is able to use the same TCP socket APIs as well as communicate with the non-MPTCP enabled hosts. MPTCP is included in the mainline Linux Kernel [MPLINUX]. It has also been used in various devices such as iPhone [I-PHONE] to improve the reliability by concurrently connecting the WiFi and the cellular interfaces. Currently, MPTCP has implemented four path managers to effectively use multiple paths between the source and the destination.

- o Default: This path manager does nothing more than accepting the passive creation of subflows.
- o Full Mesh: This creates a full-mesh of subflows between all available source and destination interfaces. Usually the subflows are created by the client.
- o Ndiffports: This path manager creates multiple subflows between every source and destination pair.

o Binder: This is a relatively newer path manager included in the list. It is based on Loose Source Routing [BOC2013].

MPTCP is completely backward compatible with the TCP socket APIs. However, these socket APIs do not exploit the complete potential of MPTCP as these functionality are not available in TCP [RFC6824]. Hence, it is not possible to provide fine grained control over MPTCP and in some ways it restricts the usability of MPTCP. Since, defining new system call is a big ask for MPTCP and application developers are more used to controlling the behavior of TCP using socket option APIs (setSockOpt and getSockopt), it is quite effective to develop new socket APIs for MPTCP as well to control its behavior.

In their draft [HESID17], Hesmans et al. have proposed MPTCP socket APIs to interact with the underlying socket and control the subflow behavior e.g. getting the list of existing subflows, creation of new subflow, termination of subflows etc.. In this draft, we use their draft [HESID17] as the foundation and provide the extended socket APIs to allow the application to mark a particular subflow as an active or a backup subflow. Such a control is very helpful for the applications which intend to send multiple data streams with different Quality of Service (QoS) [SHA2017] requirements to another application or end host.

By default MPTCP creates every new subflow as an active subflow. Hence, if a backup or low priority subflow gets disconnected and connects again, it becomes active i.e. it does not remember its previous state. In this document we also provide a new data structure to remember the states of subflows. Hence, if a particular subflow between any source destination pair is restored after disconnection, it restores with the same state.

The rest of the document is organized as follows: it gives the details of the socket APIs and the corresponding getSockOpt and setSockopt details. We than provide the details of data structure to remember the subflow priority and the corresponding socket APIs and the socket options.

2. Socket APIs for subflow Priority

In this section we present the new socket APIs for the subflow priority control. Currently, MPTCP provides two possible priorities for each subflow; high priority or active subflow, and low priority or backup subflow. These socket APIs can enable the client or the server to dynamically set the subflow priority.

For all socket APIs, the underlying connections are assumed to Multipath TCP connections, otherwise these APIs return an error and set the errono as "EOPNOTSUPP".

2.1. Control Subflow Priority

This document addresses a typical requirement for an application to change the priority of a subflow without restarting the entire connection.

The first important step to set the subflow priority is to get the list of available subflows. This can be performed using the "MPTCP_GET_SUB_IDS" socket option defined by Hensman et al.[HESID17, HES2017].

The next step is to set the priority of a particular subflow. The "MPTCP_SET_SUB_PRIO" socket option can be used to set the subflow priority identified by its subflow id. This can be called by using "setsockopt" with "MPTCP_SET_SUB_PRIO" as the socket option and passing a pointer to "mptcp_sub_prio" structure. The "mptcp_sub_prio" structure is defined as follows:

```
struct mptcp_sub_prio {
       __u8     id;
       __u16    low_prio;
   };
```

Here, "id" is the subflow id as returned by the get subflow socket API and "low_prio" is priority value to be defined for the subflow. Note that a subflow is an active subflow if the low_prio flag is set to "0". A typical illustration of this API to set the sublfow priority as backup is as follows:

```
struct mptcp_sub_prio fp = {5, 1}; //subflow id 5 is set to backup.

setsockopt(sockfd, IPPROTO_TCP, MPTCP_SET_SUB_PRIO,&fp, sizeof(fp));
```

On successful return of the above socket API, MPTCP protocol sends this information to remote host using MP_PRIO flag.

2.2. Remembering Subflow Priority

A new subflow in MPTCP is in active state and immediately starts sending the data. If the subflow between a particular source and destination is marked as backup using "MPTCP_SET_SUB_PRIO", it becomes active if the subflow gets disconnected and connects again or a new subflow is created. To handle this situation, two new lists "ActiveInterfaceList" and "BackupInterfaceList" are included in MPTCP to remember the state of the subflows between two end hosts. To resolve any inconsistency between the two lists, the former has been assigned a higher priority than the later.

To populate the active and backup lists, setsockopt with "MPTCP_SUB_PATH_ACTIVE_LIST" and "MPTCP_SUB_PATH_BACKUP_LIST" respectively can be used. This requires the application to pass the pointer to "MPTCP_SUB_PATH" structure. The "MPTCP_SUB_PATH" is defined as follows:

```
struct mptcp_sub_path {
   sa_family_t sa_family;
   union {
      struct in_addr sin_addr;
      struct in6_addr sin6_addr;
   };
   union {
      struct in_addr din_addr;
      struct in6_addr din6_addr;
   };
};
```

These lists can be updated any time; however the runtime update will not change the state of an existing subflow. Hence, this socket APIs must be called before the subflows are created.

3. Security Considerations

There are no new security considerations for this document.

4. IANA Considerations

There are no IANA considerations in this document.

5. Conclusions

This document provides extended socket APIs for MPTCP to control the subflow priorities. These are expected to be very useful for those applications which want a fine grained control over the data to be sent over the available subflows between the end hosts. These APIs can increase the versatility of MPTCP subflows and also provide an opportunity to the application developers to select the subflows more intelligently. This is expected to be useful for different scenarios and devices e.g. drones [SHA2017, RAO2017] where it is important to segregate control data from user data on different subflow.

6. References

6.1. Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC793, DOI 10.17487/RFC0793, September 1981, <http://www.rfc-editor.org/info/rfc793>.

[RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure,"TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <http://www.rfc-editor.org/info/rfc6824>.

[HESID17] B. Hesmans and O. Bonaventure, "A socket API to control Multipath TCP", Internet Draft, July 2017, https://tools.ietf.org/html/draft-hesmans-mptcp-socket-02.

[MPLINUX] "Multipath TCP implementation in the Linux kernel", <http://www.multipath-tcp.org>.

6.2. Informative References

[RFC6182] Ford, A., Raiciu, C., Handley, M., Barre, S., and J. Iyengar, "Architectural Guidelines for Multipath TCP Development", RFC6182, DOI 10.17487/RFC6182, March 2011, <http://www.rfc-editor.org/info/rfc6182>.

[I-PHONE] Apple Inc, "iOS - Multipath TCP Support in iOS7", <https://support.apple.com/en-us/HT201373>.

[BOC2013] L. Boccassi, M. M. Fayed, and M. K. Marina, "Binder: A System to Aggregate Multiple Internet Gateways in Community Networks," in Proceedings of the 2013 ACM MobiCom Workshop on Lowest Cost Denominator Networking for Universal Access. ACM, 2013, pp. 3-8.

[SHA2017] S. Shailendra, K. Aniruddh, B. Panigrahi, and A. Simha, "Multipath TCP Path Scheduler for Drones - A Segregation of Control and User Data," in Proceedings of the 18th ACM International Symposium on Mobile Ad Hoc Networking and Computing. ACM, 2017.

[RAO2017] A. Rao, M. Visali, S. Shailendra, B. Panigrahi, and A. Simha, "Reliable Robotic Communication using MPTCP," in 9th International Conference on Communication Systems and Networks, Jan 2017, pp. 429-430.

[HES2017] B. Hesmans and O. Bonaventure, "An enhanced socket API for Multipath TCP," in Proceedings of the 2016 Applied Networking Research Workshop. ACM, 2016, pp. 1-6.

7. Authors' Addresses

Samar Shailendra
TCS Research & Innovation
Bangalore, India

Email: s.samar@tcs.com

Hemant Kumar Rath
TCS Research & Innovation
Bhubaneswar, India

Email: Hemant.rath@tcs.com

Arpan Pal
TCS Research & Innovation
Kolkata, India

Email: arpan.pal@tcs.com

Abhijit Mondal
Indian Institute of Technology Kharagpur
Kharagpur, India

Email: abhimondal@iitkgp.ac.in

MPTCP
Internet Draft
Intended status: Informational
Expires: September 6, 2018

J. Zhu
W. Liu
J. Zuo
Huawei

March 5, 2018

A Path-aware Scheduling Scheme for Multipath Transport Protocols
draft-zuo-mptcp-scheduler-00.txt

Abstract

The design of scheduling data amongst multiple asynchronous paths impacts the performance of multipath transport protocols. This draft proposes a path-aware scheduling scheme, which adaptively selects the most suitable scheduling scheme according to the path characteristics in a time-variant multipath transport scenario. When the path characteristics are unknown, the redundant mode is employed to achieve at least as good performance as that of the best single path. The condition for choosing the most suitable scheduling mode relies on the measured path characteristics and the application requirements. Especially, the path-aware scheduling scheme is designed in this draft in terms of of the delay difference of multiple paths, compared with a pre-defined delay threshold.

Status of this Memo

Copyright and License Notice

Table of Contents

1. Introduction

A suitable scheduling mode will influence the performance of multiple transport protocols[RFC6824] due to the asynchronous path characteristics. When the delays of the multiple paths differ, the packets with consecutive sequence number will arrive at the receiver at different time, causing the out-of-order problem. Some pre-scheduled schemes are proposed to pre-schedule the sending time of each packet ensuring they are delivered in order. Some other scheduling solutions are trying to re-inject the data in flight from the slow subflow to the fast subflow and reduce the cwnd size of the slow subflow[OppRetr]. However, all these methods require the knowledge of the accurate path characteristics. If the path characteristics are unknown, the scheduling schemes are arbitrary, while if the network environment varies violently, the scheduling schemes can not respond to the variant network in time and may become invalid.

The redundant scheduling mode[MPTCP.org] is useful if the latency is the primary goal, which will try to transmit the traffic on all available subflows in a redundant way no matter what the path characteristics is. However, it achieves the low latency by sacrificing the bandwidth. The min_RTT scheduling mode is the best known up to today, which will always first send data on the subflow with the lowest RTT until the cwnd is full. However, it depends on the accurate path characteristics measurement. If there is no data sent on one of the subflows, it can not get the subflow's current path characteristics.

Therefore, a new path-aware scheduling scheme for multiple transport protocols is proposed to adaptively select the suitable scheduling mode according to the path characteristics. Specifically, if the characteristics of all paths are unknown, the redundant mode is the appropriate method to send data in all paths concurrently. Meanwhile, the path characteristics are measured during the redundant transmission in all the paths. Once the path characteristics are obtained, the scheduling mode is adaptively selected according to the measured path characteristics and the application requirements. This draft also defines the scheduling thresholds to judge the suitable scheduling mode at a certain scenario. The thresholds could be defined as the delay difference, the rate, etc., where their value could be decided by the experience or given by the applications requirements.

It is intended that the scheduling scheme presented in this draft can be applied to other multipath transport protocols, such as alternative multipath extensions to TCP[RFC793], UDP, QUIC, or indeed any other multipath protocols. However, for the purposes of example, this document will, where appropriate, refer to the MPTCP[RFC6182].

2. Acronyms and 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 RFC 2119 [RFC2119].

3. A New Path-Aware Scheduling Scheme

The new path-aware scheduling scheme adaptively selects the most suitable scheduling mode according to the measured path characteristics compared with a pre-defined threshold. The redundant mode is employed when the path characteristics are unknown. In this draft, we take the latency-sensitive traffic as an example, which considers the delay difference as the mode-selecting condition.

3.1. The Redundant/Non-Redundant modes

Two modes are defined: 1) The redundant mode sends the same data in all paths concurrently; 2)The non-redundant mode sends data in min_RTT mode or just the best single path.

3.2. The Mode-Selecting Condition

For latency-sensitive traffic, the delay is the most important factor to make a scheduling decision. Hence, a delay difference is defined as the mode-selecting condition. Due to the asynchronous path characteristics, the RTT can not really reflect the delay of the data delivered to the receiver, the one-way-latency[OWL] is suggested as the condition, where the receiver calculates the delay with the arriving time and its corresponding sending time, shown as

one-way latency (i) = receiving time (i) - sending time (i) .

The one-way latency is fed back to the sender carried in the one-way latency (MP_OWL) Option, then the sender can calculate the delay difference, as

delay difference (i) = |one-way latency (i) - one-way latency (j)|.

According to the application requirements, a threshold is pre-configured and used to compare with the measured delay difference. When the delay difference is less than the threshold, the redundant mode is employed, otherwise the non-redundant mode is used. For example, when the non-redundant mode is selected, the path with the lowest one-way latency is used to send data. If the delay difference changes until that it is larger than the threshold, then the non-redundant mode switches back to the redundant mode.

3.3. The Path Characteristics Measurement

As described above, the one-way latency is calculated at the receiver and fed back to the sender in the option of the acknowledgement packet, while the delay difference is calculated at the sender. If no data is scheduled in a certain path for a long time, it can not get the one-way latency in time, and the scheduling mode may become invalid. The phenomenon of no data scheduled in a certain path usually happens when the delay difference of the multiple paths is large while with limited receive buffer. It also happens when the application traffic generates in burst intermittently.

During the initialization stage, the path characteristics are unknown, hence the redundant mode is used to measure the one-way latency. Afterwards, if the selected scheduling mode only sends data in a certain path according to the measured delay difference, two alternative methods are designed to make the sender obtain the one-way latency of the other path, which are 1) Periodically switch to the redundant mode, which transmits the same data in all paths and measure the one-way latency; 2) Periodically send a signalling packet in the other path to measure the one-way latency, which reduce the network overhead compared to the method 1.

4. Implementation Considerations

As described in Section 3.2, the delay difference is calculated according to the one-way latency of multiple paths. The one-way latency is carried in a new defined option, the one-way latency (MP_OWL) Option, and sent back to the sender. The format is shown as follows, where the value of the Subtype could be defined as '0x8'.

```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-----------------------+
|     Kind      |    Length     |Subtype| (reserved)            |
+---------------+---------------+-------+-----------------------+
|        One-way latency(4/8 octets, depending on flags)        |
+---------------------------------------------------------------+
```

Figure 1: One-way Latency (MP_OWL) Option

5. Security Considerations

TBD.

7. Acknowledgements

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

8.2. Informative References

[RFC793] Postel, J., "Transmission Control Protocol", STD 7, RFC793, DOI 10.17487/RFC0793, September 1981, <http://www.rfc-

editor.org/info/rfc793>.

[RFC6824] Ford, A., Raiciu, C., Handley, M., and Bonaventure, O., "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <http://www.rfc-editor.org/info/rfc6824>.

[RFC6182] Ford, A., Raiciu, C., Handley, M., Barre, S., Iyengar, J., "Architectural Guidelines for Multipath TCP Development", RFC 6182, DOI 10.17487/RFC6182, March 2011, <http://www.rfc-editor.org/info/rfc6182>.

[MPTCP.org] Multipath TCP - Linux Kernel Implementation, <http://multipath-tcp.org/pmwiki.php/Users/ConfigureMPTCP>.

[OppRetr] Raiciu, C., Paasch, C., Barre, S., Ford, A., Honda, M., Duchene, F., Bonaventure, O. and Handley, M., "How hard can it be? designing and implementing a deployable multipath TCP", April 2012, In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (pp. 29-29).

Author’s Addresses

Jianjian Zhu
Huawei Technologies
Bantian, Longgang District,
Shenzhen 518129 P.R. China
EMail: zhujianjian1@huawei.com

Wei Liu
Huawei Technologies
Bantian, Longgang District,
Shenzhen 518129 P.R. China
EMail: liuwei57@huawei.com

Jing Zuo
Huawei Technologies
Bantian, Longgang District,
Shenzhen 518129 P.R. China
EMail: jing.zuo@huawei.com