Multipath TCP
Internet-Draft
Intended status: Standards Track
Expires: December 29, 2016

R. Barik
University of Oslo
S. Ferlin
Simula Research Laboratory
M. Welzl
University of Oslo

June 27, 2016

A Linked Slow-Start Algorithm for MPTCP
draft-barik-mptcp-lisa-01

Abstract

This document describes the LISA (Linked Slow-Start Algorithm) for Multipath TCP (MPTCP). Currently during slow-start, subflows behave like independent TCP flows making MPTCP unfair to cross-traffic and causing more congestion at the bottleneck. This also yields more losses among the MPTCP subflows. LISA couples the initial windows (IW) of MPTCP subflows during the initial slow-start phase to remove this adverse behavior.

Status of this Memo

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

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

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

This Internet-Draft will expire on December 29, 2016.

Copyright Notice

Table of Contents

1. Introduction

The current MPTCP implementation provides multiple congestion control algorithms, which aim to provide fairness to TCP flows at the shared bottlenecks. However, in RFC 6356 [RFC6356], the subflows' slow-start phase remains unchanged to RFC 5681 [RFC5681], and all the subflows at this stage behave like independent TCP flows. Following the development of IW as per [RFC6928], each MPTCP subflow can start with IW = 10. With an increasing number of subflows, the subflows' collective behavior during the initial slow-start phase can temporarily be very aggressive towards a concurrent regular TCP flow at the shared bottleneck.

According to [UIT02], most of the TCP sessions in the Internet consist of short flows, e.g., HTTP requests, where TCP will likely never leave slow-start. Therefore, the slow-start behavior becomes of critical importance for the overall performance.

To mitigate the adverse effect during initial slow-start, we introduce LISA, the "Linked Slow-Start Algorithm". LISA shares the congestion window MPTCP subflows in slow start whenever a new subflow joins.

1.1. Definitions

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].

Acronyms used in this document:

IW -- Initial Window

RTT -- Round Trip Time

CWND -- Congestion Window

Inflight -- MPTCP subflow's inflight data

old_subflow.CWND -- Congestion Window of the subflow having largest sending rate

new_subflow.CWND -- New incoming subflow's Congestion Window

Ignore_ACKs -- a boolean variable indicating whether ACKs should be ignored

ACKs_To_Ignore -- the number of ACKs for which old_subflow.CWND stops increasing during slow-start

compound CWND -- sum of CWND of the subflows in slow-start

2. MPTCP Slow-Start Problem Description

Since it takes 1 RTT for the sender to receive any feedback on a given TCP connection, sending an additional segment after every ACK is rather aggressive. Therefore, in slow-start, all subflows independently doubling their CWND as in regular TCP results in MPTCP also doubling its compound CWND. The MPTCP aggregate only diverges from this behavior when the number of subflows changes. Coupling of CWND is therefore not necessary in slow-start except when a new subflow joins.

2.1. Example of current MPTCP slow-start problem

We illustrate the problematic MPTCP slow-start behavior with an example: Consider an MPTCP connection consisting of 2 subflows. The first subflow starts with IW = 10, and after 2 RTTs the CWND becomes 40 and a new subflow joins, again with IW = 10. Then, the compound CWND becomes 40+10 = 50. With an increasing number of subflows, the compound CWND in MPTCP becomes larger than that of a concurrent TCP flow.

For example, MPTCP with eight subflows (as recommended in [DCMPTCP11] for datacenters) will have a compound CWND of 110 (40+7*10). As a result, MPTCP would behave unfairly to a concurrent TCP flow sharing the bottleneck. This aggressive behavior of MPTCP also affects the performance of MPTCP. If multiple subflows share a bottleneck, each of them doubling their rate every RTT, will cause excessive losses at the bottleneck. This makes MPTCP enter the congestion avoidance phase earlier and thereby increases the completion time of the transfer.

This problem, and the improvement attained with LISA, are documented in detail in [lisa].

3. Linked Slow-Start Algorithm

3.1. Description of LISA

The idea behind LISA is that each new subflow takes a ’packet credit’ from an existing subflow in slow-start for its own IW. We design the mechanism such that a new subflow has 10 segments as the upper limit

[RFC6928] and 3 segments as the lower limit [RFC3390]. This is based on [RFC6928], [RFC3390] and the main reason behind it is to let these subflows compete reasonably with other flows. We also divide the CWND fairly in order to give all subflows an equal chance when competing with each other.

LISA first finds the subflow with the largest sending rate measured over the last RTT. Depending on the subflow's CWND, between 3 and 10 segments are taken from it as packet credit and used for the new subflow's IW. The packet credit is realized by reducing the CWND from the old subflow and halting its increase for ACKs_To_Ignore number of ACKs.

We clarify LISA with the example given in Section 2.1. After 2 RTTs, the old_subflow.CWND = 40 and a new_subflow joins the connection. Since old_subflow.CWND >= 20 (refer to Section 3.2), 10 packets can be taken by the new_subflow.CWND, resulting in old_subflow.CWND = 30 and new_subflow.CWND = 10. Hence, MPTCP's compound CWND, whose current size is 40, should ideally become 60+20 = 80 after 1 RTT (assuming a receiver without delayed ACKs). However, if 40 segments from old_subflow.CWND are already in flight, the compound CWND becomes in fact 70+20 = 90. Here, LISA keeps old_subflow.CWND from increasing for the next 10 ACKs. In comparison, MPTCP without LISA would have a compound CWND of 80+20=100 after 1 RTT.

3.2. Algorithm

Below, we describe the LISA algorithm. LISA is invoked before a new subflow sends its IW.

1. Before computing the new_subflow.CWND, Ignore_ACKs = False and ACKs_To_Ignore = 0.

2. Then, ignoring the new_subflow, the subflow in slow-start with the largest sending rate (old_subflow.CWND, measured over the last RTT) is selected.

3. If there is no such subflow, the IW of the new_subflow.CWND = 10 Otherwise, the following steps are executed:

```
if old_subflow.CWND >= 20 // take IW(10) packets

   old_subflow.CWND -= 10

   new_subflow.CWND = 10
```

```
            Ignore_ACKs = True
         else if old_subflow.CWND >= 6 // take half the packets
            new_subflow.CWND -= old_subflow.CWND / 2
            old_subflow.CWND -= new_subflow.CWND
            Ignore_ACKs = True
         else
            new_subflow.CWND = 3 // can't take from old_subflow
   4.  if Ignore_ACKs and Inflight > old_subflow.CWND
         // do not increase CWND when ACKs arrive
         ACKs_To_Ignore = Inflight - old_subflow.CWND
```

4. Implementation Status

LISA is implemented as a patch to the Linux kernel 3.14.33+ and within MPTCP's v0.89.5. It is meant for research and provided by the Unviersity of Oslo and Simula Research Laboratory, and available for download from http://heim.ifi.uio.no/runabk/lisa This code was used to produce the test results that are reported in [lisa].

5. Acknowledgements

This work was part-funded by the European Community under its Seventh Framework Programme through the Reducing Internet Transport Latency (RITE) project (ICT-317700). The authors also would like to thank David Hayes (UiO) for his comments. The views expressed are solely those of the authors.

6. IANA Considerations

This memo includes no request to IANA.

7. Security Considerations

8. Change History

Changes made to this document:

00->01 : Some minor text improvements and updated a reference.

9. References

9.1. 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>.

[RFC3390] Allman, M., Floyd, S., and C. Partridge, "Increasing TCP's Initial Window", RFC 3390, DOI 10.17487/RFC3390, October 2002, <http://www.rfc-editor.org/info/rfc3390>.

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

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

[RFC6928] Chu, J., Dukkipati, N., Cheng, Y., and M. Mathis, "Increasing TCP's Initial Window", RFC 6928, DOI 10.17487/RFC6928, April 2013, <http://www.rfc-editor.org/info/rfc6928>.

9.2. Informative References

[DCMPTCP11] Raiciu, C., Barre, S., Pluntke, C., Greenhalgh, A., Wischik, D., and M. Handley, "Improving datacenter performance and robustness with multipath TCP", ACM SIGCOMM p266-277, August 2011.

[UIT02] Brownlee, N. and K. Claffy, "Understanding internet traffic streams: Dragonflies and tortoises", IEEE Communications Magazine p110-117, 2002.

[lisa] Barik, R., Welzl, M., Ferlin, S., and O. Alay, "LISA: A Linked Slow-Start Algorithm for MPTCP", IEEE ICC 2016,

Kuala Lumpur, Malaysia , 2016.

Authors' Addresses

Runa Barik
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Email: runabk@ifi.uio.no

Simone Ferlin
Simula Research Laboratory
P.O.Box 134
Lysaker, 1325
Norway

Email: ferlin@simula.no

Michael Welzl
University of Oslo
PO Box 1080 Blindern
Oslo, N-0316
Norway

Phone: +47 2285 2420
Email: michawe@ifi.uio.no

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: January 5, 2017

M. Boucadair
C. Jacquenet
Orange
D. Behaghel
OneAccess
S. Secci
UPMC
W. Henderickx
Nokia/Alcatel-Lucent
R. Skog
Ericsson
O. Bonaventure
Tessares
S. Vinapamula
Juniper
S. Seo
Korea Telecom
W. Cloetens
SoftAtHome
U. Meyer
Vodafone
LM. Contreras
Telefonica

July 4, 2016

An MPTCP Option for Network-Assisted MPTCP Deployments: Plain Transport Mode
draft-boucadair-mptcp-plain-mode-08

Abstract

Because of the lack of Multipath TCP (MPTCP) support at the server side, some service providers now consider a network-assisted model that relies upon the activation of a dedicated function called MPTCP concentrator. This document focuses on a deployment scheme where the identity of the MPTCP concentrator(s) is explicitly configured on connected hosts.

This document specifies an MPTCP option that is used to avoid the encapsulation of packets and out-of-band signaling between the CPE and the MPTCP concentrator. Also, this document specifies how UDP traffic, in particular, can be distributed among available paths by leveraging MPTCP capabilities.

Requirements Language

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

Status of This Memo

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

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

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

This Internet-Draft will expire on January 5, 2017.

Copyright Notice

Table of Contents

1. Introduction

One of the promising deployment scenarios for Multipath TCP (MPTCP, [RFC6824]) is to enable a Customer Premises Equipment (CPE) that is connected to multiple networks (e.g., DSL, LTE, WLAN) to optimize the usage of such resources. This deployment scenario is called a network-assisted MPTCP model, and relies upon MPTCP proxies located on both the CPE and network sides (Figure 1). The latter plays the role of an MPTCP concentrator. Such concentrator terminates the MPTCP sessions established from CPEs, before redirecting traffic into legacy TCP sessions.

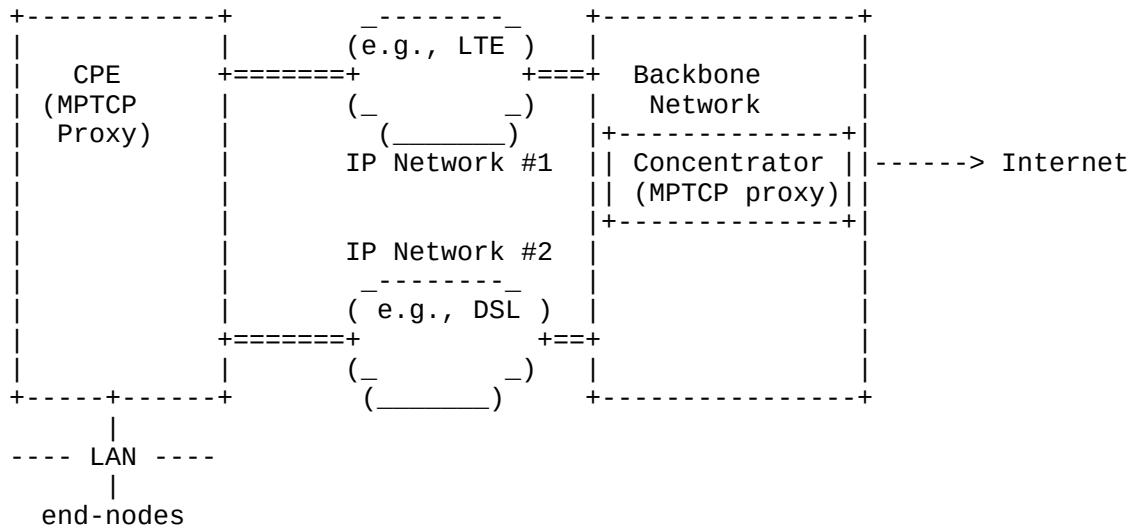
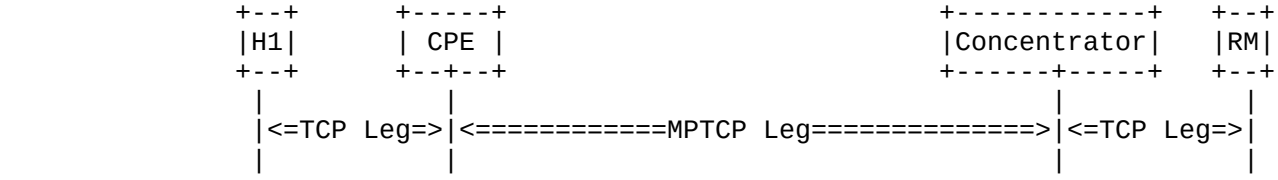


Figure 1: "Network-Assisted" MPTCP Design

Network-assisted MPTCP deployment models are designed to facilitate the adoption of MPTCP for the establishment of multi-path communications without making any assumption about the support of MPTCP by the communicating peers. Thus, MPTCP proxies deployed in CPEs and in concentrators located in the network are responsible for establishing multi-path communications on behalf of endpoints, thereby taking advantage of MPTCP capabilities to optimize resource usage to achieve different goals that include (but are not limited to) bandwidth aggregation, primary/backup communication paths, and traffic offload management. Figure 2 depicts the various TCP connection legs in network-assisted MPTCP deployment models.



Legend:
H1: Host 1
RM: Remote Machine

Figure 2: Connection Legs (CPE-based Model)

There are also MPTCP deployments to assist hosts that are directly connected to multiple networks to establish multi-path

communications. The communication legs that are involved in such deployments are shown in Figure 2.

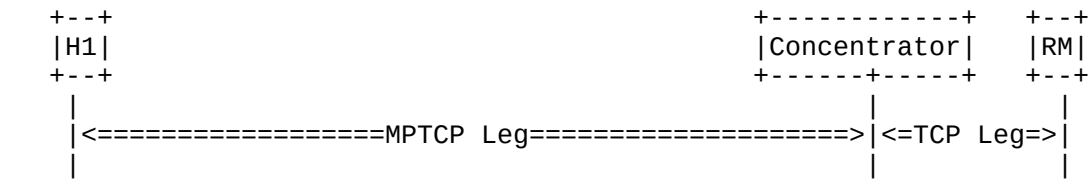


Figure 3: Connection Legs (Host-based Model)

Most of the current operational deployments that take advantage of multi-interfaced devices rely upon the use of an encapsulation scheme (such as GRE [I-D.zhang-gre-tunnel-bonding]). The use of encapsulation is motivated by the need to steer traffic towards the concentrator and also to allow the distribution of any kind of traffic besides TCP (e.g., UDP) among the available paths without requiring any advanced traffic engineering tweaking technique in the network side to intercept traffic and redirect it towards the appropriate concentrator.

This specification assumes an MPTCP concentrator is reachable by means of one or multiple IP addresses. Also, it assumes the various network attachments provided to an MPTCP-enabled device (CPE or host) are managed by the same administrative entity. The IP reachability information of an MPTCP concentrator can be explicitly configured on a device, e.g., by means of a specific DHCP option [I-D.boucadair-mptcp-dhc]. This document assumes such explicit configuration. Additional assumptions are listed in Section 3.

Current operational MPTCP deployments by network operators are focused on the forwarding of TCP traffic. In addition, the design of such deployments sometimes assumes the use of extra signalling provided by SOCKS [RFC1928], at the cost of additional management complexity and possible service degradation (e.g., up to 8 SOCKS messages may need to be exchanged between two MPTCP proxies before an MPTCP connection is established, thereby yielding several tens of milliseconds of extra delay before the connection is established) .

To avoid the burden of encapsulation and additional signalling between MPTCP proxies, this document explains how a plain transport mode is enabled, so that packets are exchanged between the CPE and the concentrator without requiring the activation of any encapsulation scheme (e.g., IP-in-IP [RFC2473], GRE [RFC1701]). This plain transport mode also avoids the need for out-of-band signalling.

The solution described in this document also works properly when NATs are present in the communication path between the CPE and the concentrator, unlike solutions that rely upon GRE tunneling. In particular, the solution proposed in this document accommodates deployments that involve CGN (Carrier Grade NAT) upstream the concentrator.

The plain transport mode is characterized as follows:

- No encapsulation required (no tunnels, whatsoever).
- No out-of-band signaling for each MPTCP subflow required.
- Carries any protocol (incl. UDP) for the benefit of massive MPTCP adoption (Section 5).
- Accommodates various deployment contexts (Section 6).

2. Terminology

The reader should be familiar with the terminology defined in [RFC6824].

This document makes use of the following terms:

Customer-facing interface: is an interface of the MPTCP concentrator that is visible to a CPE or a host directly connected to the operator's network, and which is used for communication purposes between a CPE/host and the MPTCP concentrator.

Internet-facing interface: is an interface of the MPTCP concentrator that is visible to a remote host on the Internet.

IP transport address: refers to an IP address and transport port number.

MPTCP proxy: is a software module that is responsible for transforming a TCP connection into an MPTCP connection, and vice versa. Typically, an MPTCP proxy is embedded in a CPE and a concentrator.

MPTCP leg: refers to a network segment where MPTCP is used to establish TCP connections (see Figure 2).

MPTCP concentrator (or concentrator): refers to a functional element that is responsible for aggregating traffic pertaining to a group of CPEs. This element is typically located upstream in the network, e.g., beyond a Broadband Network Gateway (BNG) or a PDN Gateway (PGW) in wired and wireless access network environments, respectively. One or multiple concentrators can be deployed in

the network to help MPTCP-enabled CPEs establish MPTCP connections via available network attachments.

On the uplink path, the concentrator terminates the MPTCP connections received from its customer-facing interfaces and transforms these connections into legacy TCP connections towards upstream servers.

On the downlink path, the concentrator converts the legacy server's TCP connections into MPTCP connections towards its customer-facing interfaces.

3. Assumptions & Scope

The following assumptions are made:

- The logic for mounting network attachments by a CPE (or a host directly connected to the operator's network) is deployment- and implementation-specific and is out of scope of this document.
- Policies can be enforced by a concentrator instance operated by the Network Provider to manage both upstream and downstream traffic. These policies may be subscriber-specific, connection-specific, system-wide, or else.
- The concentrator may be notified about monitoring results (e.g., provided by passive or active probes) that detail the status of the various network legs available to service a customer, a group of customers, a whole region, etc. No assumption is made in this document about how these monitoring operations are executed.
- An MPTCP-enabled, multi-interfaced CPE or host that is directly connected to one or multiple access networks is allocated addresses/prefixes via legacy mechanisms (e.g., DHCP) supported by the various available network attachments. The CPE/host may be assigned the same or distinct IP address/prefix via the various available network attachments.
- The location of the concentrator(s) is deployment-specific. Network Providers may choose to adopt centralized or distributed designs. Nevertheless, in order to take advantage of MPTCP, the location of the concentrator should not jeopardize packet forwarding performance overall.
- The logic of traffic distribution over multiple paths is deployment-specific. This document does not require nor preclude any particular traffic distribution schemes.

- o No assumption is made whether one single or multiple IP addresses/ prefixes are assigned to host connected to a CPE.

It is out of the scope of this document to discuss criteria for selecting traffic to be eligible to MPTCP service. It is out of scope of the document to specify how a CPE selects its concentrator(s), too.

Likewise, methods to avoid TCP fragmentation, such as rewriting the TCP Maximum Segment Size (MSS) option, are out of scope for this document.

This document focuses on the CPE-based model (i.e., the CPE embeds a MPTCP proxy that behaves on behalf of terminal devices), but plain transport mode can also apply to host-based models.

TCP/MPTCP session tracking by the MPTCP proxy is implementation-specific. Readers may refer to Section 2 of [RFC7857].

This specifications focuses on TCP and UDP. Future documents may specify the exact behavior for transporting other protocols over MPTCP connections.

Also, this specification focuses on a stateful design; stateless approaches that rely on including the Plain Mode option in all packets are out of scope.

4. Plain Transport Mode Behavior

As shown in Figure 2, TCP connections initiated by a host are converted by the CPE into MPTCP connections towards the concentrator. Then, the concentrator converts these connections into legacy TCP connections towards the final destinations. Since the concentrator can be located anywhere in the operator’s network, Section 4.1 introduces a new TCP option to supply the concentrator with required information to forward the traffic to its final destination. When a CPE receives a SYN segment from a host of the LAN, it rewrites the destination address of that segment to an address of the concentrator, and places the original destination (and possibly source) addresses in this TCP option. Further details are specified in the following sub-sections.

Specific UDP processing is discussed in Section 5.

4.1. Plain Mode MPTCP Option

The Plain Mode (PM) option carries the source/destination IP addresses and/or port numbers of the origin source and destination nodes. It is also used to indicate whether the data carried in the packet is relayed from a native TCP connection or refers to the use of another transport protocol. The format of the option is shown in Figure 4.

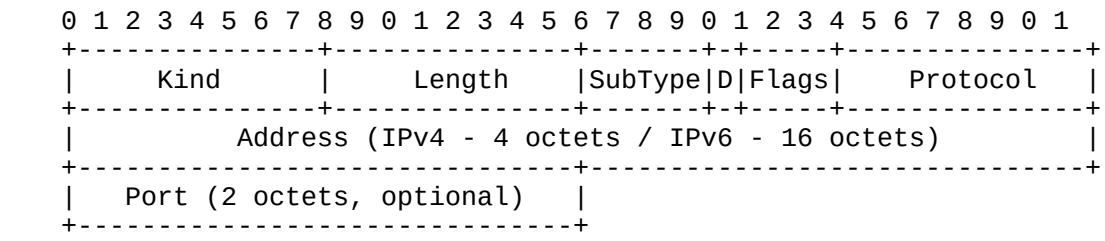


Figure 4: Plain Mode MPTCP Option

The description of the fields is as follows:

- Kind and Length: are the same as in Section 3 of [RFC6824].
- Subtype: to be defined by IANA (Section 8). Implementations may use "0xe" subtype encoding for early deployment purposes in managed networks.
- D-bit (direction bit): this flag indicates whether the enclosed IP address (and port number) reflects the source or destination IP address (and port). When the D-bit is set, the enclosed IP address must be interpreted as the source IP address. When the D-bit is unset, the enclosed IP address must be interpreted as the destination IP address.
- "Flag" bits: are reserved bits for future assignment as additional flag bits. These additional flag bits MUST each be set to zero and MUST be ignored upon receipt.
- Protocol: conveys the protocol number as assigned by IANA [proto_numbers]. For example, this field is set to 17 for UDP traffic, or 6 for TCP. The processing of UDP flows is further discussed in Section 5.
- Address: includes a source or destination IP address. The address family is determined by the "Length" field. Concretely, a PM option containing an IPv4 address has a Length field of 8 bytes (or 10 if a port number is included). A PM option containing an

IPv6 address has a Length of 20 bytes (or 22 if a port number is included).

- Port: If the D-bit is set (resp. unset), a source (resp. destination) port number may be associated with the IP address. This field is valid for protocols that use a 16 bit port number (e.g., UDP, TCP, SCTP).

4.2. Carrying the Plain Mode Option

When using an MPTCP connection to forward traffic (whether it's TCP traffic or any other traffic), the CPE (resp. the concentrator) MUST insert a Plain Mode option in a SYN packet sent to the concentrator (resp. the CPE). The Plain Mode option MUST be included in the SYN payload.

Note: Given the length of the PM option, especially when IPv6 addresses are used, and the set of TCP options that are likely to be included in a SYN message, it will not always be possible to place the PM option inside the dedicated TCP option space. Given that this option is designed to be used in a controlled environment, this specification recommends to always place the PM options inside the payload of a SYN segment. Including data in a SYN payload is allowed as per Section 3.4 of [RFC0793].

If the original SYN message contains data in its payload (e.g., [RFC7413]), that data MUST be placed right after PM and "End of Options List" (EOL) options when generating the SYN in the MPTCP leg. The EOL option serves as a marker to delineate the end of the TCP options and the beginning of the data included in the original SYN .

The Plain Mode option MUST only appear in SYN segments that contain the MP_CAPABLE option. SYN messages to create subsequent subflows of a given MPTCP connection MUST NOT include any PM option (Figure 5).

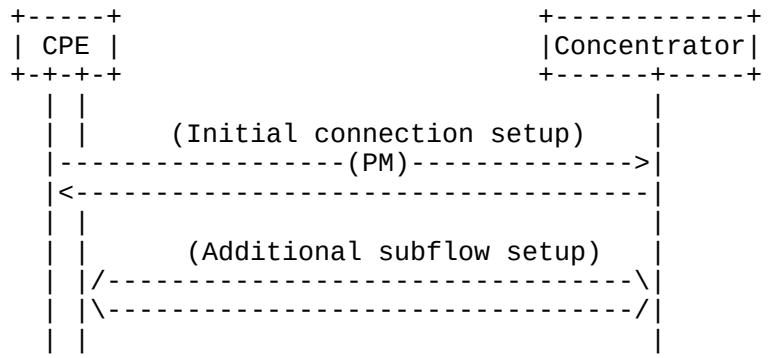


Figure 5: Carrying the Plain Mode Option (Focus on the MPTCP Leg)

By default, source IP address preservation is assumed for IPv6 while global address sharing is assumed for IPv4. This means that, by default, two plain mode option instances MUST be included in a SYN segment for IPv6 (both source and destination) and one instance MUST be present for IPv4 (either the source or the destination). The CPE and the concentrator MUST support a configurable parameter to modify this default behavior to accommodate alternate deployment models (see Section 6).

An implementation MUST ignore PM options that include multicast, broadcast, and host loopback addresses [RFC6890].

The 'Protocol' field of the PM option MUST be copied from the 'Protocol' field of the IPv4 header or set to the type of the transport header of the IPv6 packet that will be forwarded along MPTCP subflows. The CPE and the concentrator MAY be configured to disable traffic aggregation for some transport protocols because of the nature of the service they relate to (e.g., IP multicast traffic typically specific of live TV broadcasting services). By default, TCP and UDP traffic bonding MUST be enabled.

4.3. Binding Tables

4.3.1. On the Need to Maintain a State

Because the source IP and/or destination IP addresses are communicated only during the SYN exchange of the initial subflow, the CPE and the concentrator MUST maintain a state that binds the MPTCP transport coordinates to the destination/source IP address, ports, and protocol. This specification discusses the external behavior of this stateful design; the internal behavior for maintaining that state is implementation-specific.

This document uses 'Internal transport session identifier' to identify a particular transport session on the LAN side of the CPE and 'External transport session identifier' to identify a particular transport session on the Internet-facing Interface of the concentrator. An implementation could use the classical 4-tuple (source and destination addresses and ports) as such an identifier.

An MPTCP proxy also needs to identify a particular MPTCP connection. We refer to it as the 'MPTCP transport coordinates'. An implementation could, for example, use the token assigned to a specific connection to identify an MPTCP connection. The 4-tuple of each subflow that belong to an MPTCP connection can also be part of the MPTCP transport coordinates.

Binding entries can be created as a result of a packet or be configured directly on the CPE or the concentrator.

4.3.2. Binding & Transport Session Entries

An implementation may maintain distinct binding tables, each for a given transport protocol, or maintain one single binding table to handle all supported transport protocols.

Subflows can be added or deleted during the lifetime of an MPTCP connection based on triggers that are local to the CPE/concentrator, based on signals received from the concentrator/CPE, or as a result of processing a packet. These triggers are outside the scope of this specification.

The CPE must maintain a binding entry that allows to associate the internal transport address (IP address, port number) with one or a set of external IP transport addresses, that are assigned in the WAN interfaces of the CPE in the context of a given MPTCP connection. Each of the external transport addresses points to a subflow that is created between the CPE and the concentrator. For each binding entry, one or multiple transport session entries are maintained by the CPE and the concentrator. These session entries are meant to store the information that is required for rewriting packets. A session entry is created as a result of handling a packet.

A session entry maintained by the CPE may be structured as follows:

Internal transport session identifier: This information typically include the source IP transport address (IPl, Pl) and the destination IP transport address (IPd, Pd) of the connection.

MPTCP transport coordinates: These coordinates include information about the subflows that compose this MPTCP connection. When a

packet matches an existing binding entry, the CPE may decide whether existing subflows can be used to forward the packet, or whether a new subflow is to be created.

The following information is maintained for each MPTCP subflow:

* (IPwi, Pwi): The source IP address and port for this subflow.

* (IPci, Pci): IPci is one of the concentrator's IP addresses, while Pci is a port number selected to establish this subflow. This information is used as the destination IP address and port of a packet matching this entry.

Transport protocol: This information is typically retrieved from the outgoing packet that will be placed in MPTCP connections. The transport protocol specifies the protocol that is used in the LAN side.

Lifetime: This information indicates the remaining validity lifetime for the session entry. When the lifetime expires, this session entry is deleted from the table. If all sessions bound to a given binding entry expired, that binding entry must be deleted. Recommendations for setting this parameter are defined in [RFC7857][RFC5382][RFC4787].

For example:

o An outgoing packet {src=(IPl, Pl); dst=(IPd,Pd)} will be transformed by the CPE to {src=(IPwi,Pwi); dst=(IPci,Pci)}.

o An incoming packet {src=(IPci,Pci); dst=(IPwi,Pwi) will be transformed by the CPE to {src=(IPd, Pd); dst=(IPl,Pl)};

The structure of a session entry maintained by the concentrator may be as follows:

External transport session identifier: This information typically include the external IP transport address (IPe, Pe) and the destination IP transport address (IPd, Pd) of the connection. The external IP transport address is set to the (IPl, Pl) if and only if the concentrator is configured to preserve the source IP address and port. In such case, this information is retrieved from the PM option included in a SYN packet. Otherwise, the external IP address and port are selected by the concentrator from a local pool.

MPTCP transport coordinates: These coordinates include information about the subflows that compose this MPTCP connection. When a

packet matches an existing binding entry, the concentrator may decide whether existing subflows can be used to forward the packet, or whether a new subflow is to be created.

The following information is maintained for each MPTCP subflow:

* (IPwi, Pwi): The source IP address and port for this subflow.

* (IPci, Pci): The destination IP address and port for this subflow. It can be set by the CPE or selected by the concentrator.

Transport protocol: This information is retrieved from the PM option included in a SYN packet. The transport protocol specifies the protocol that must be used when sending the packet through the Internet-facing interface.

Lifetime: This information indicates the remaining validity lifetime for the session entry. When the lifetime expires, this session entry is deleted from the table. If all sessions bound to a given binding entry expired, that binding entry must be deleted. Recommendations for setting this parameter are defined in [RFC7857][RFC5382][RFC4787].

For example:

o An outgoing packet {src=(IPwi,Pwi); dst=(IPci,Pci)} will be transformed by the concentrator to {src=(IPe,Pe); dst=(IPd,Pd)}.

o An incoming packet {src=(IPd,Pd); dst=(IPe,Pe) will be transformed by the concentrator to {src=(IPd, Pd); dst=(IPci,Pci)}.

4.3.3. Expiration of a Binding Entry

A configurable parameter MAY be supported by the CPE and the concentrator to terminate MPTCP connections with the FASTCLOSE procedure (Section 3.5 of [RFC6824]) when a binding entry expires.

If there is no binding state that matches a received non-SYN segment, the CPE/concentrator SHOULD reply with a RST segment. This behavior aims to synchronize the binding tables between the CPE and the concentrator by clearing bindings that are present either in the CPE or in the concentrator.

The configurable parameter is set by default to ’Disable’.

4.4. Theory of Operation: Focus on TCP

4.4.1. Processing an Outgoing SYN

PM option usage for an outgoing TCP SYN (i.e., from the CPE to the concentrator) is as follows:

(1) Outgoing TCP SYNs that can be forwarded by a CPE along MPTCP subflows are transformed by the CPE into TCP packets carried over an MPTCP connection.

The decision-making process to decide whether a given flow should be MPTCP-serviced or not is local to the CPE, and reflects the service-inferred policies as defined by the bonding service provider. As such, the decision-making process is policy-driven, implementation- and deployment-specific.

(2) As a result, SYNs packets are sent over an MPTCP connection according to the plain transport mode (i.e., without any encapsulation header), and the related instructions carried in the PM option.

The source IP address and port number are those assigned to one of the CPE WAN interfaces. Because multiple IP addresses may be available to the CPE, the address used to rewrite the source IP address for an outgoing packet forwarded through a given network attachment (typically, a WAN interface) MUST be associated with that network attachment. It is assumed that ingress traffic filtering policies ([RFC2827]) are enforced at the network boundaries to prevent any spoofing attack.

The destination IP address is replaced by the CPE with one of the IP addresses of the concentrator.

The destination port number may be maintained as initially set by the host or altered by the CPE.

The original destination and/or source IP address are copied into Plain Mode options. The option is inserted as per the guidelines documented in Section 4.2.

A session entry (including the protocol) MUST be maintained by the CPE for that outgoing packet (Section 4.3). A timeout is associated with this entry as per the recommendations in [RFC5382].

(3) Upon receipt of a SYN packet on its MPTCP leg, the concentrator extracts the IP address(es) included in the PM option and uses

it as the destination (and possibly the source) IP address of the corresponding SYN packet that it will forward towards its final destination. The 'Protocol' field of the PM option indicates the transport protocol that must be used when sending the packet through the Internet-facing interface.

The source IP address and port belong to a pool that is configured to the concentrators if address or prefix rewriting is enabled (see Section 6). A session entry MUST be instantiated by the concentrator to record the state (see Section 4.3).

The concentrator may be configured to behave as either a 1:1 IPv4 address translator or a N:1 IPv4 address translator where a given global IPv4 address is therefore shared by multiple CPEs. Network Providers should be aware of the complications that may arise if a given IP address/prefix is shared by multiple customers (see [RFC6269][RFC6967]). Whether these complications apply or not to a network-assisted MPTCP environment is deployment-specific.

The concentrator should preserve the same external IP address that was assigned to a given CPE for all its outgoing connections when forwarding traffic from an MPTCP connection to the Internet (i.e., use an "IP address pooling" behavior of "Paired") [RFC4787]. The port allocation policy configured on the concentrator (e.g., port set assignment, deterministic NAPT, etc.) is implementation and deployment-specific.

4.4.2. Processing an Incoming SYN

In order to appropriately handle incoming SYN packets, the concentrator (resp. CPE) are supposed to be configured with instructions that allows to redirect the traffic to the appropriate CPE (resp. Internal host).

Plain transport mode operation for an incoming TCP SYN (i.e., when traffic is forwarded from the concentrator towards the CPE) is as follows:

(1) If the incoming TCP SYN matches a binding entry (Section 4.3), the concentrator rewrites some of the packet's fields according to the information maintained in this entry. In addition, the concentrator records the source IP address and port in the PM option. Also, the 'Protocol' field of the PM option is set according to the guidelines in Section 4.2.

The source IP address is replaced with one of the IP addresses listed in the binding information base maintained by the concentrator.

The destination IP address is replaced with one of the CPE's IP addresses.

A session entry is instantiated to record the transport-related information to rewrite the packet.

(2) Upon receipt of the TCP SYN by the CPE, it extracts the IP address included in the Plain Mode option and uses it as the source IP address of the packet that the CPE will forward through its LAN interface until the packet reaches its final destination.

The destination IP address, port, and protocol are retrieved from a binding entry maintained by the CPE.

4.4.3. Processing Subsequent Outgoing/Incoming Non-SYNs

The required information to rewrite non-SYN packets that match an existing binding entry, is retrieved from the Binding Information Bases (BIB) maintained by the CPE and the concentrator (see Section 4.3). The MPTCP proxy may decide at any time to create or terminate subflows associated to an MPTCP connection. When a packet arrives, its content is transported over one of the subflows of a bound MPTCP connection.

Non-SYN messages exchanged in the context of an existing subflow and all messages for non-initial subflows do not include the PM option.

4.4.4. Handling TCP RST Messages

RST messages may be received from the LAN side of the CPE or by the concentrator in its Internet-facing interface. When the CPE or the concentrator receive a TCP RST matching an existing entry, it MUST apply the FASTCLOSE procedure defined in Section 3.5 of [RFC6824]) to terminate the MPTCP connection and the associated subflows. The transport coordinates of the FASTCLOSE messages are set according to the information maintained in the binding table.

The CPE and the concentrator SHOULD wait for 4 minutes before deleting the session and removing any state associated with it if no packets are received during that 4-minute timeout [RFC7857].

5. Processing UDP Traffic

This document leverages the ability to create MPTCP connections on the CPE/concentrator to also carry data conveyed in UDP datagrams. A UDP flow can be defined as a series of UDP packets that have the same source and destination address and ports. Upon receipt of the first packet of such a flow, a binding entry (Section 4.3) is created to map this flow onto an MPTCP connection between the CPE and the concentrator. All the subsequent UDP segments of this UDP flow are transported over that MPTCP connection. The MPTCP connection is released when no traffic is exchanged for this flow (Section 5.1.3).

5.1. Behavior

From an application standpoint, there may be a value to distribute UDP datagrams among available network attachments for the sake of network resource optimization, for example. This document uses MPTCP features to control how UDP datagrams are distributed among existing network attachments. The data carried in UDP datagrams belonging to a given UDP flow are therefore transported in an MPTCP connection. An MPTCP connection is bound to one UDP flow. New MPTCP connections are created in order to handle additional UDP flows.

The management of MPTCP connections that are triggered by UDP datagrams follows the guidelines documented in [RFC6824].

The following sub-sections exclusively focus on the external behavior to achieve UDP to TCP conversion (Section 5.1.1), and vice versa (Section 5.1.2).

5.1.1. UDP to TCP Conversion

This function is applied to UDP traffic received by the CPE from the LAN, and to UDP traffic received by the concentrator from one of its Internet-facing interfaces.

When the CPE (or the concentrator) receives a UDP datagram to be distributed over MPTCP subflows, it MUST check whether the packet matches an existing binding entry (Section 4.3).

If an entry is found, and the packet is to be placed on an existing subflow, the packet is processed according to the corresponding session entry. If an entry is found, but the packet should be placed on a new subflow, a session entry MUST be instantiated by the CPE for that outgoing packet. The information about the transport protocol (UDP, in this case) MUST also be included in this binding entry. In both cases, the CPE (or the concentrator) MUST proceed as follows:

1. Extract the payload and its length from the UDP datagram.

2. Send the length (as a 16 bits field in network byte order) followed by the payload of the UDP datagram over the bound MPTCP connection.

UDP packets that are received by the concentrator, but do not match an existing binding, MUST be silently dropped.

UDP packets that are received by the CPE, but do not match an existing binding, MUST be proceed as follows:

1. Instantiate a new binding entry for this outgoing packet. The information about the transport protocol (UDP, in this case) MUST also be included in this binding entry.

2. Initiate the MPTCP connection that will be used to carry the UDP datagrams of this flow towards the chosen concentrator. For this, the CPE MUST create a SYN segment containing the following information :

   * The MP_CAPABLE option and possibly other TCP options.

   * The payload contains the following information (in this order):

     + A PM option indicating the original source address and port if source address preservation is enabled.

     + A PM option indicating the original destination address and port.

     + The EOL TCP option.

     + The Length of the UDP payload in network byte order.

     + The payload of the UDP datagram.

When setting the source IP address, the destination IP address, and the IP address enclosed in the Plain Mode MPTCP option of the corresponding TCP packet, the same considerations as specified in Section 4.3 MUST be applied.

Whether one or multiple UDP payloads are included in the same TCP segment is implementation- and deployment-specific.

5.1.2. TCP to UDP Conversion

Upon receipt of a SYN segment containing the PM option specifying the UDP protocol, the concentrator MUST proceed as follows:

- Create a binding entry to map this MPTCP connection to a UDP flow (Section 4.3).
- Extract the destination, and possibly source, transport addresses from the PM option and complete the session entry with this information.
- Extract the UDP payload.
- Generate a UDP datagram with the corresponding IP addresses and ports and the UDP payload.

Upon receipt of a SYN segment containing the PM option specifying the UDP protocol, the CPE MUST proceed as follows:

- If no binding is found, the packet MUST be silently dropped.
- If a binding is found:
  - Extract the source (optionally, destination) transport addresses from the PM option.
  - Create a session entry to map this MPTCP connection to a UDP flow (Section 4.3).
  - Extract the UDP payload.
  - Generate a UDP datagram with the corresponding IP addresses and ports and the UDP payload.

Upon receipt of data over an MPTCP connection that is bound to a UDP flow, the 'Length' field is used to extract the UDP payloads from the bytestream and generates the corresponding UDP datagrams.

The concentrator (or the CPE) MUST follow the same procedure as mentioned in Section 4.3 for address and port rewriting purposes.

5.1.3. Terminating UDP-Triggered Subflows

UDP-triggered subflows SHOULD be terminated by an MPTCP endpoint (CPE or concentrator) if no UDP packet matching the corresponding binding entry is received for at least 5 minutes (see Section 4.3 of [RFC4787]). Consequently, the procedure in Section 4.4.4 MUST be

followed to terminate the MPTCP connection and the associated subflows. The transport coordinates of the FASTCLOSE messages are set according to the information maintained in the binding table.

5.2. Examples

A flow example is shown in Figure 6 to illustrate how TCP packets are generated to relay UDP datagrams using several subflows. Non-SYN messages that belong to a given subflow do not include any PM option. Also, this example shows how subsequent UDP datagrams of this flow are transported over the existing subflow or how a new subflow is created. In this example, the SYN segment issued to add a new subflow also includes data received in the original UDP datagram.

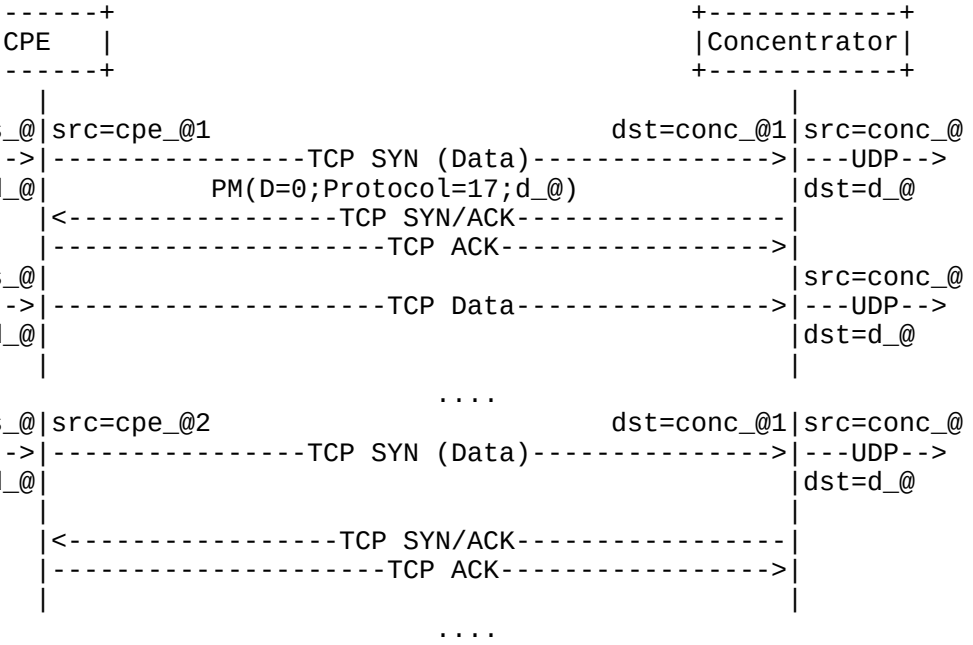


Figure 6: Distributing UDP packets over multiple paths (1)

Figure 7 shows an example of UDP datagrams that are transported over MPTCP subflows. Unlike the previous example, additional subflows to transport UDP datagrams of the same flow are established in advance between the CPE and the concentrator.

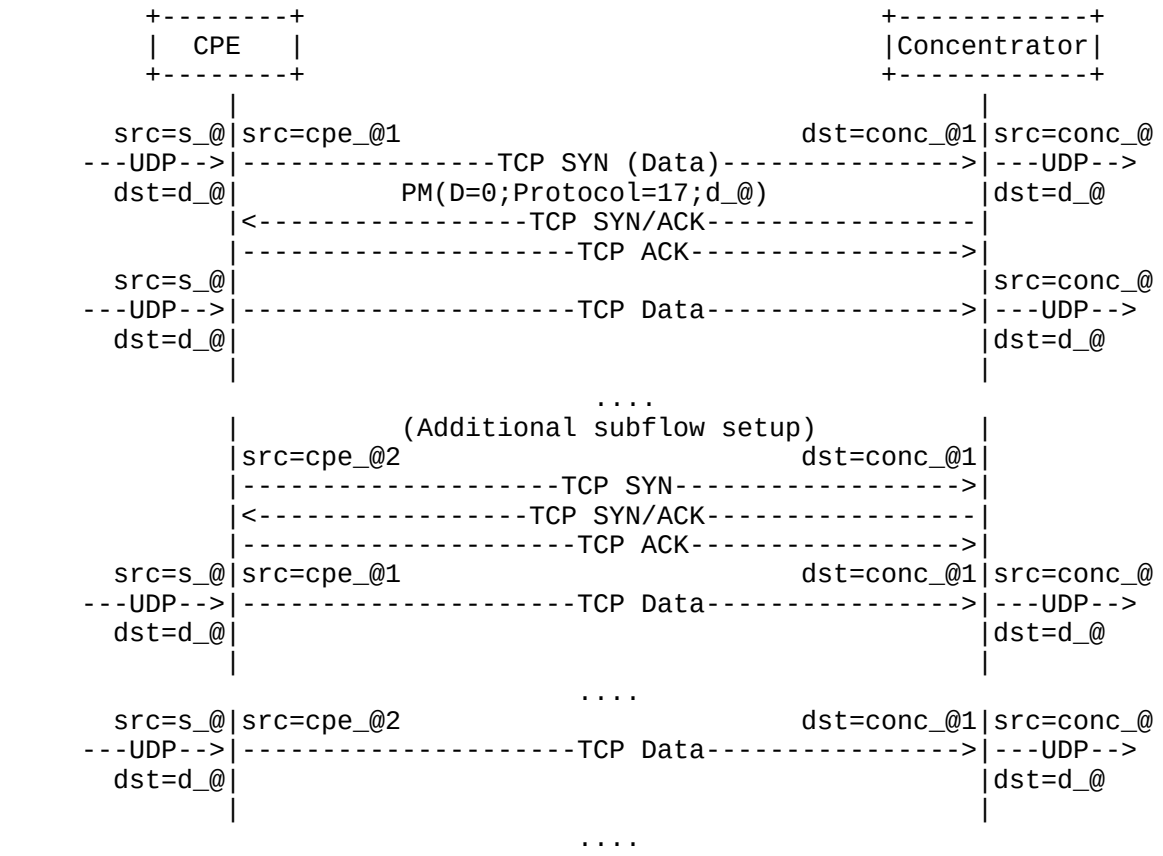


Figure 7: Distributing UDP packets over multiple paths (2)

5.3. Fragmentation & Reassembly Considerations

The subsequent UDP/TCP header swapping introduced in Section 5.1 represents an overhead that is equal to the difference between TCP and UDP header sizes. To avoid fragmentation when processing large UDP datagrams, it is RECOMMENDED to increase the MTU of all links between the CPE and the concentrator to accommodate this overhead.

Nevertheless, in deployments where increasing the MTU of all links is not possible for some reason, the CPE and the concentrator SHOULD be configurable to enable/disable fragmentation and reassembly of UDP datagrams. The decision to enable or disable this parameter is deployment-specific. This parameter is set to 'Disabled' by default.

If this configurable parameter is set to 'Disabled', large UDP datagrams that may thus be fragmented MUST NOT be forwarded along the

MPTCP connection, i.e., the bonding service MUST NOT be applied to such large packets.

If this configurable parameter is set to 'Enabled', the CPE and the concentrator MUST perform IPv4 fragmentation and reassembly for packets that exceed the link MTU. Concretely, IPv4 fragmentation MUST be performed once UDP/TCP header swapping is completed. Packet reassembly MUST occur before TCP/UDP header swapping. The behavior to adopt whenever the swapping of UDP/TCP headers leads to IPv4 fragmentation is as follows:

- Present the packet to the MPTCP proxy as per Section 5.1.1.
- Fragment the transformed packet (TCP), and then forward the fragments.

The remote MPTCP endpoint (CPE or concentrator) then adopts the following behavior:

- Reassemble the TCP packet,
- Present the packet to the MPTCP proxy as per Section 5.1.2.

In order to protect the CPE and the concentrator and minimize the risk of degrading the overall bonding service performance, dedicated resources SHOULD be reserved for handling fragments (e.g., by limiting the amount of resources to process out-of-order packets).

5.3.1. Receiving IPv4 Fragments on the Internet-Facing Interface of the Concentrator

The forwarding of an IPv4 packet received on the Internet-facing interface of the concentrator requires the IPv4 destination address and the transport-protocol destination port for binding lookup purposes. If the first packet received contains the transport-protocol information, the concentrator uses a cache and forwards the fragments unchanged (i.e., without reassembly). A description of such a caching algorithm is outside the scope of this document. If subsequent fragments arrive before the first fragment, the concentrator SHOULD queue these fragments till the first fragment is received.

The processing of the first fragment MUST follow the same procedure as in Section 5.1.1. The rewriting of the IP addresses of subsequent fragments MUST follow the instructions maintained in the binding table and the fragmentation cache. The MF (More Fragments) bit and 'Fragment offset' field MUST NOT be modified by the concentrator.

5.3.2. Receiving IPv4 Fragments from the LAN

If the first packet received contains the transport-protocol information, the CPE uses a cache and forwards the fragments unchanged (i.e., without reassembly). If subsequent fragments arrive before the first fragment, the concentrator SHOULD queue these fragments till the first fragment is received.

The processing of the first fragment MUST follow the same procedure as in Section 5.1.2. The rewriting of the IP addresses of subsequent fragments MUST follow the instructions maintained in the binding table and the fragmentation cache. The MF bit and 'Fragment offset' field MUST NOT be modified by the CPE.

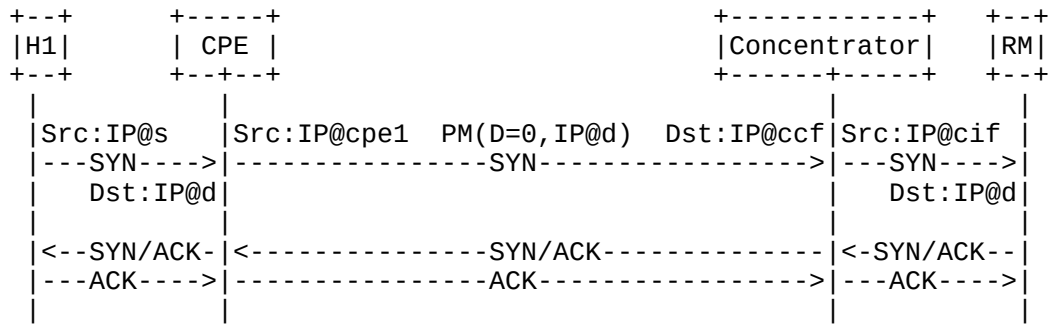
5.3.3. Distinct Address Families

If distinct address families are used in the UDP and MPTCP legs, fragmentation SHOULD be handled as described in Sections 4 and 5 of [RFC7915].

6. Deployment Scenarios

The Plain Transport Mode accommodates various deployment contexts such as:

IPv4 address sharing: Because of global IPv4 address depletion, optimization of the IPv4 address usage is mandatory, and this includes IPv4 addresses that are assigned by the concentrator at its Internet-facing interfaces (Figure 8). A pool of global IPv4 addresses is provisioned to the concentrator along with possible instructions about the address sharing ratio to apply (see Appendix B of [RFC6269]). Adequate forwarding policies are enforced so that traffic destined to an address of such pool is intercepted by the appropriate concentrator.



Legend:
ccf: Concentrator Customer-facing Interface
cif: Concentrator Internet-facing Interface

Figure 8: Example of Outgoing SYN without Source Address Preservation

IPv4 address 1:1 translation: For networks that do not face global IPv4 address depletion yet, the concentrator can be configured so that source IPv4 addresses of the CPE are replaced with other (public) IPv4 address. A pool of global IPv4 addresses is then provisioned to the concentrator for this purpose. Rewriting source IPv4 addresses may be used as a means to redirect incoming traffic towards the appropriate concentrator.

Source IPv6 address preservation: Some IPv6 deployments may require the preservation of the source IPv6 address (Figure 9). This model avoids the need for the concentrator to support ALGs to accommodate applications with IPv6 address referrals. In order to intercept incoming traffic, specific IPv6 routes are injected so that traffic is redirected towards the concentrator.

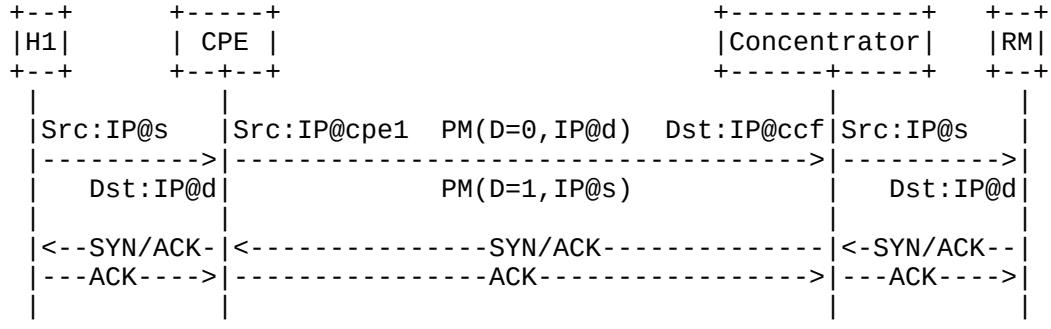


Figure 9: Example of Outgoing SYN with Source Address Preservation

IPv6 prefix sharing (NPTv6): Rewriting the source IPv6 prefix ([RFC6296]) may be needed to redirect incoming traffic towards the appropriate concentrator. A pool of IPv6 prefixes is then provisioned to the concentrator for this purpose.

Subflows of a given MPTCP connection can be associated to the same address family or may be established with different address families. Also, the plain transport mode applies regardless of the addressing scheme enforced by each CPE network attachment. In particular, the plain transport mode indifferently accommodates the following combinations.

LAN Leg	CPE-Concentrator Legs	Concentrator-RM Leg
IPv4	IPv4	IPv4
IPv4	IPv6	IPv4
IPv4	IPv6 & IPv4	IPv4
IPv6	IPv6	IPv6
IPv6	IPv4	IPv6
IPv6	IPv6 & IPv4	IPv6

Also, the CPE and the concentrator may be configured to preserve the same DSCP marking or enforce DSCP re-marking policies, and the plain transport mode described in this document fully respects these DSCP marking policies. Those considerations are deployment-specific.

7. Additional Considerations

7.1. Authorization

The Network Provider that manages the various network attachments (including the concentrators) may enforce authentication and authorization policies using appropriate mechanisms. For example, a non-exhaustive list of methods to achieve authorization is provided hereafter:

- The network provider may enforce a policy based on the International Mobile Subscriber Identity (IMSI) to verify that a user is allowed to benefit from the aggregation service. If that authorization fails, the PDP context /bearer won’t be mounted. This method does not require any involvement from the concentrator.

- The network provider may enforce a policy based on Access Control Lists (ACLs), e.g., at the Broadband Network Gateway (BNG) to control the CPEs that are authorized to communicate with a concentrator. These ACLs may be installed as a result of RADIUS

exchanges, for instance. This method does not require any involvement from the concentrator.

- The concentrator may implement an Ident interface [RFC1413] to retrieve an identifier that will be used to assess whether that client is authorized to make use of the aggregation service. Ident exchanges will take place only when receiving the first subflow from a given source IP address.

- The concentrator may embed a RADIUS client that will solicit an AAA server to check whether connections received from a given source IP address are authorized or not.

A first safeguard against the misuse of the concentrator resources by illegitimate users (e.g., users with access networks that are not managed by the same operator owning the concentrator) is to reject MPTCP connections received on the Internet-facing interfaces. Only MPTCP connections received on the customer-facing interfaces of a concentrator will be accepted.

Because only the CPE is entitled to establish MPTCP connections with a concentrator, ACLs may be installed on the CPE to avoid that internal terminals issue MPTCP connections towards one of the concentrators.

7.2. Checksum Adjustment

Given that the TCP and UDP checksum covers the pseudo-header that contains the source and destination IP addresses, the checksum should be updated to reflect the change of these addresses. For the particular case of UDP/TCP conversion (Section 5), the UDP checksum can be computed from the TCP one and vice versa.

7.3. Logging

If the concentrator is used in global IPv4 address sharing environments, the logging recommendations discussed in Section 4 of [RFC6888] need to be considered. Security-related issues encountered in address sharing environments are documented in Section 13 of [RFC6269].

7.4. Middlebox Interference

The use of the Plain Transport Mode option is primarily meant for MPTCP designs that involve access networks managed by the same operator. Appropriate setup is required before MPTCP with the Plain Transport Mode option is activated, so that possible middleboxes

located in these access networks do not strip MPTCP signals, nor remove data contained in the SYN payload.

The plain transport mode may be deployed at large but some complications may arise, e.g., if an in-path middlebox removes the MPTCP option or data from the SYN payload. These complications not specific to the Plain Mode, and are encountered whenever MPTCP is deployed.

7.5. EPC Billing & Accounting

In case that one of MPTCP subflow between CPE and concentrator includes mobile (e.g., LTE, 3G, etc), billing and accounting of the traffic may be considered per subflow, per subscriber, or else. Since packets generated from/to the subscriber (CPE) are destined/ sourced to/from the concentrator, the EPC nodes may need to inspect, in some deployments, the destination/source address and/or port included in the plain mode option to check and make billing and accounting actions. Alternate deployment approaches may be adopted to avoid inspecting L3/4 information (e.g., rely on application-based filters, correlate flow characteristics retrieved using Policy and Charging Control (PCC) interfaces, etc.).

It is out of the scope of this document to make any recommendation in that area.

8. IANA Considerations

This document requests an MPTCP subtype code for this option:

- Plain Mode MPTCP Option

  NOTE: Implementations may use "0xe" subtype encoding for early deployment purposes in managed networks.

9. Security Considerations

MPTCP-related security threats are discussed in [RFC6181] and [RFC6824]. Additional considerations are discussed in the following sub-sections.

9.1. Privacy

The concentrator may have access to privacy-related information (e.g., IMSI, link identifier, subscriber credentials, etc.). The concentrator MUST NOT leak such sensitive information outside a local domain.

9.2. Denial-of-Service (DoS)

Means to protect the MPTCP concentrator against Denial-of-Service (DoS) attacks MUST be enabled. Such means include the enforcement of ingress filtering policies at the network boundaries [RFC2827].

In order to prevent the exhaustion of concentrator's resources, by establishing a large number of simultaneous subflows for each MPTCP connection, the administrator SHOULD limit the number of allowed subflows per CPE for a given connection. Means to protect against SYN flooding attacks MUST also be enabled ([RFC4987]).

Attacks that originate outside of the domain can be prevented if ingress filtering policies are enforced. Nevertheless, attacks from within the network between a host and a concentrator instance are yet another actual threat. Means to ensure that illegitimate nodes cannot connect to a network should be implemented.

9.3. Illegitimate Concentrator

Traffic theft is a risk if an illegitimate concentrator is inserted in the path. Indeed, inserting an illegitimate concentrator in the forwarding path allows traffic intercept and can therefore provide access to sensitive data issued by or destined to a host. To mitigate this threat, secure means to discover a concentrator should be enabled.

9.4. High Rate Reassembly

The CPE and the concentrator may perform packet reassembly. Some security-related issues are discussed in [RFC4963][RFC1858][RFC3128].

10. Acknowledgements

Many thanks to Chi Dung Phung, Mingui Zhang, Rao Shoaib, Yoshifumi Nishida, and Christoph Paasch for the comments.

Thanks to Ian Farrer, Mikael Abrahamsson, Alan Ford, Dan Wing, and Sri Gundavelli for the fruitful discussions in IETF#95 (Buenos Aires).

Special thanks to Pierrick Seite, Yannick Le Goff, Fred Klamm, and Xavier Grall for their valuable comments.

Thanks also to Olaf Schleusing, Martin Gysi, Thomas Zasowski, Andreas Burkhard, Silka Simmen, Sandro Berger, Michael Melloul, Jean-Yves Flahaut, Adrien Desportes, Gregory Detal, Benjamin David, Arun Srinivasan, and Raghavendra Mallya for the discussion.

11. References

11.1. Normative References

[proto_numbers] http://www.iana.org/assignments/protocol-numbers, "Protocol Numbers".

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

[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>.

[RFC4787] Audet, F., Ed. and C. Jennings, "Network Address Translation (NAT) Behavioral Requirements for Unicast UDP", BCP 127, RFC 4787, DOI 10.17487/RFC4787, January 2007, <http://www.rfc-editor.org/info/rfc4787>.

[RFC5382] Guha, S., Ed., Biswas, K., Ford, B., Sivakumar, S., and P. Srisuresh, "NAT Behavioral Requirements for TCP", BCP 142, RFC 5382, DOI 10.17487/RFC5382, October 2008, <http://www.rfc-editor.org/info/rfc5382>.

[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>.

[RFC6890] Cotton, M., Vegoda, L., Bonica, R., Ed., and B. Haberman, "Special-Purpose IP Address Registries", BCP 153, RFC 6890, DOI 10.17487/RFC6890, April 2013, <http://www.rfc-editor.org/info/rfc6890>.

[RFC7857] Penno, R., Perreault, S., Boucadair, M., Ed., Sivakumar, S., and K. Naito, "Updates to Network Address Translation (NAT) Behavioral Requirements", BCP 127, RFC 7857, DOI 10.17487/RFC7857, April 2016, <http://www.rfc-editor.org/info/rfc7857>.

[RFC7915] Bao, C., Li, X., Baker, F., Anderson, T., and F. Gont, "IP/ICMP Translation Algorithm", RFC 7915, DOI 10.17487/RFC7915, June 2016, <http://www.rfc-editor.org/info/rfc7915>.

11.2. Informative References

[I-D.boucadair-mptcp-dhc]
Boucadair, M., Jacquenet, C., and T. Reddy, "DHCP Options for Network-Assisted Multipath TCP (MPTCP)", draft-boucadair-mptcp-dhc-05 (work in progress), May 2016.

[I-D.zhang-gre-tunnel-bonding]
Leymann, N., Heidemann, C., Zhang, M., Sarikaya, B., and M. Cullen, "Huawei's GRE Tunnel Bonding Protocol", draft-zhang-gre-tunnel-bonding-03 (work in progress), May 2016.

[RFC1413] St. Johns, M., "Identification Protocol", RFC 1413, DOI 10.17487/RFC1413, February 1993, <http://www.rfc-editor.org/info/rfc1413>.

[RFC1701] Hanks, S., Li, T., Farinacci, D., and P. Traina, "Generic Routing Encapsulation (GRE)", RFC 1701, DOI 10.17487/RFC1701, October 1994, <http://www.rfc-editor.org/info/rfc1701>.

[RFC1858] Ziemba, G., Reed, D., and P. Traina, "Security Considerations for IP Fragment Filtering", RFC 1858, DOI 10.17487/RFC1858, October 1995, <http://www.rfc-editor.org/info/rfc1858>.

[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, <http://www.rfc-editor.org/info/rfc1928>.

[RFC2473] Conta, A. and S. Deering, "Generic Packet Tunneling in IPv6 Specification", RFC 2473, DOI 10.17487/RFC2473, December 1998, <http://www.rfc-editor.org/info/rfc2473>.

[RFC2827] Ferguson, P. and D. Senie, "Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing", BCP 38, RFC 2827, DOI 10.17487/RFC2827, May 2000, <http://www.rfc-editor.org/info/rfc2827>.

[RFC3128] Miller, I., "Protection Against a Variant of the Tiny Fragment Attack (RFC 1858)", RFC 3128, DOI 10.17487/RFC3128, June 2001, <http://www.rfc-editor.org/info/rfc3128>.

[RFC4963] Heffner, J., Mathis, M., and B. Chandler, "IPv4 Reassembly Errors at High Data Rates", RFC 4963, DOI 10.17487/RFC4963, July 2007, <http://www.rfc-editor.org/info/rfc4963>.

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

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

[RFC6269] Ford, M., Ed., Boucadair, M., Durand, A., Levis, P., and P. Roberts, "Issues with IP Address Sharing", RFC 6269, DOI 10.17487/RFC6269, June 2011, <http://www.rfc-editor.org/info/rfc6269>.

[RFC6296] Wasserman, M. and F. Baker, "IPv6-to-IPv6 Network Prefix Translation", RFC 6296, DOI 10.17487/RFC6296, June 2011, <http://www.rfc-editor.org/info/rfc6296>.

[RFC6888] Perreault, S., Ed., Yamagata, I., Miyakawa, S., Nakagawa, A., and H. Ashida, "Common Requirements for Carrier-Grade NATs (CGNs)", BCP 127, RFC 6888, DOI 10.17487/RFC6888, April 2013, <http://www.rfc-editor.org/info/rfc6888>.

[RFC6967] Boucadair, M., Touch, J., Levis, P., and R. Penno, "Analysis of Potential Solutions for Revealing a Host Identifier (HOST_ID) in Shared Address Deployments", RFC 6967, DOI 10.17487/RFC6967, June 2013, <http://www.rfc-editor.org/info/rfc6967>.

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

Authors' Addresses

Mohamed Boucadair
Orange
Rennes 35000
France

Email: mohamed.boucadair@orange.com

Christian Jacquenet
Orange
Rennes
France

Email: christian.jacquenet@orange.com

Denis Behaghel
OneAccess

Email: Denis.Behaghel@oneaccess-net.com

Stefano Secci
Universite Pierre et Marie Curie
Paris
France

Email: stefano.secci@lip6.fr

Wim Henderickx
Nokia/Alcatel-Lucent
Belgium

Email: wim.henderickx@alcatel-lucent.com

Robert Skog
Ericsson

Email: robert.skog@ericsson.com

Olivier Bonaventure
Tessares
Belgium

Email: olivier.bonaventure@tessares.net

Suresh Vinapamula
Juniper
1137 Innovation Way
Sunnyvale, CA 94089
USA

Email: Sureshk@juniper.net

SungHoon Seo
Korea Telecom
Seoul
Korea

Email: sh.seo@kt.com

Wouter Cloetens
SoftAtHome
Vaartdijk 3 701
3018 Wijgmaal
Belgium

Email: wouter.cloetens@softathome.com

Ullrich Meyer
Vodafone
Germany

Email: ullrich.meyer@vodafone.com

Luis M. Contreras
Telefonica
Spain

Email: luismiguel.contrerasmurillo@telefonica.com

MPTCP Working Group
Internet-Draft
Intended status: Experimental
Expires: January 9, 2017

F. Duchene
O. Bonaventure
UCLouvain

July 08, 2016

Multipath TCP Address Advertisement
draft-duchene-mptcp-add-addr-00

Abstract

Multipath TCP [RFC6824] defines the ADD_ADDR option that allows a host to announce its addresses to the remote host. In this document we propose some improvements to this mechanism.

Status of This Memo

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

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

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

This Internet-Draft will expire on January 9, 2017.

Copyright Notice

Table of Contents

1. Introduction

Multipath TCP is an extension to TCP [RFC0793] that was specified in [RFC6824]. Multipath TCP allows hosts to use multiple paths to send and receive the data belonging to one connection. For this, a Multipath TCP is composed of several TCP connections that are called subflows. [RFC6824] defines two options to manage the host addresses:

- ADD_ADDR is used to announce one address bound to a host (possibly combined with a port number)
- REMOVE_ADDR is used to indicate that an address previously attached to a host is not anymore attached to this host

To cope with Network Address Translation (NAT), the ADD_ADDR and REMOVE_ADDR options contain an address identifier encoded as an 8 bits integer.

When the initial subflow is created, it is assumed to be initiated from the address of the client whose identifier is 0 towards the address of the server whose identifier is also 0. Both the client and the server can use ADD_ADDR to advertise the other addresses that they use. When an additional subflow is created, the MP_JOIN option placed in the SYN (resp. SYN+ACK) contains the identifier of the address used to create (resp. accept) the subflow.

The latest Multipath TCP draft [I-D.ietf-mptcp-rfc6824bis] defines the ADD_ADDR option as shown in Figure 1.

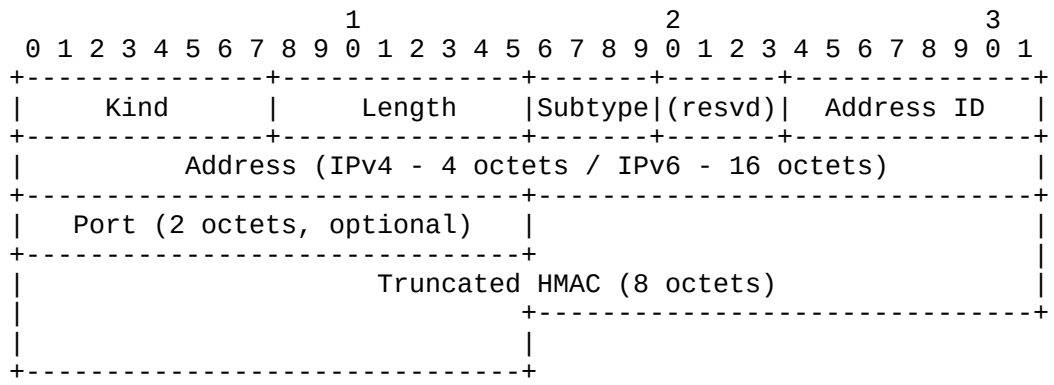


Figure 1: The Multipath TCP ADD_ADDR option format

In this document, we propose to slightly modify the format of this option based on issues that have been detected while working with the Multipath TCP implementation in the Linux kernel. More precisely, we address four different problems. The first, discussed in Section 2.1, is that the ADD_ADDR option is sent unreliably. This implies that the host sending an ADD_ADDR option cannot be sure that the address that it has advertised has been learned by the distant host. The second issue, discussed in Section 2.2 is the handling of backup subflows. Multipath TCP supports the creation of backup subflows through the B bit in the MP_JOIN option. These backup subflows consume energy and radio ressources on mobile devices and it would be useful for a host to be able to advertise a backup address that would be used to create subflows after a failure. The third issue is that multihomed hosts may have preferences on the utilisation of some of their addresses/interfaces to create additional subflows. Section 2.3 proposes a priority field that allows them to advertise these preferences. The fourth issue is that multihomed hosts, especially with IPv6, often have several addresses assigned to each interface. In this case, it can be difficult to establish disjoint paths between the communicating hosts. Section 2.4 proposes a community field in the ADD_ADDR option to indicate that some addresses share the same path. The last issue, discussed in Section 2.5 is that some hosts, e.g. servers behind a load balancer or clients behind a firewall, may want to indicate that the address used for the initial subflow should not be used to create additional ones.

2. Proposed ADD_ADDR option

To cope with the issues described later in this document we propose a new format for this option. The format for this new option is shown in Figure 2.

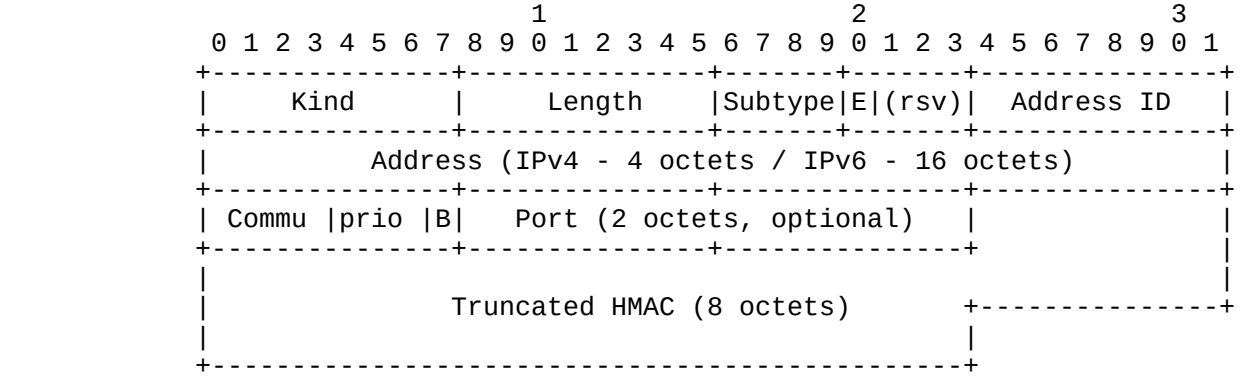


Figure 2: The proposed Multipath TCP new ADD_ADDR option format

2.1. Reliability

A first issue with the ADD_ADDR option is that since it is transmitted as a TCP option, it is not delivered reliably [Cellnet12]. When a host announces an IPv4 address, it can insert the ADD_ADDR option inside a segment that carries data that would thus be delivered reliably like user data. However, if the ADD_ADDR option contains an IPv6 address, it might be too large to fit inside a segment that already contains a DSS option and possibly other options such as the [RFC1323] timestamps. Given its length, the ADD_ADDR option cannot be placed in the same segment as a DSS option. In these two cases, the ADD_ADDR option will be often transmitted inside a duplicate ACK that is not delivered reliably. [Cellnet12] proposes a method to improve the reliability of the transmission of the ADD_ADDR option, but to our knowledge this method has never been implemented. To cope with packet losses, we propose to rely on the "E" (Echo) flag in the ADD_ADDR option (Figure 3).

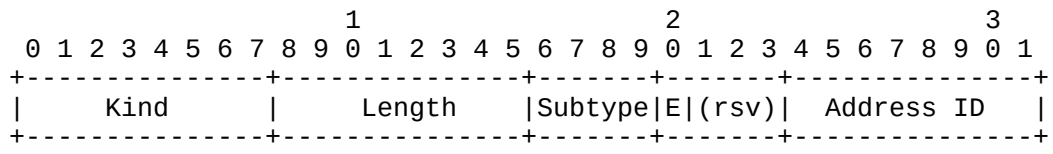


Figure 3: The part of the proposed Multipath TCP new ADD_ADDR option format with the Echo flag

The "E" flag is the "Echo" flag. When set to 0, it indicates that the host sending this option is advertising a new address to the receiving host. When set to 1, it indicates that the host sending this option acknowledges the reception of an ADD_ADDR option by echoing it. Upon reception of an ADD_ADDR option without the "E" flag set, the receiving host MUST return the exact option that it received with the "E" flag set to 1 to indicate the reception of the ADD_ADDR option. If an host advertising a new address does nt receive an echo, or receives an invalid echo of the option it MAY retransmit the ADD_ADDR. To cope with the loss of the echo of the option, if an host that advertised a new address without receiving the echo receives an MP_JOIN on this address, it MUST consider this address as having been echoed, and MUST NOT retransmit this ADD_ADDR again.

2.2. Backup

The subflows that compose a Multipath TCP connection are not all equal. [RFC6824] defines two types of subflows:

- the regular subflows
- the backup subflows

The regular subflows can be used to transport any data. The backup subflows are intended to be used only when all the regular subflows fail. [RFC6824] defines them by using the following sentence: "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."

In [RFC6824] a host can specify the type of a subflow during the three-way-handshake by using the "B" flag of the MP_JOIN option as shown in Figure 4 and Figure 5 or when the subflow is already established by sending an MP_PRIO option shown in Figure 6.

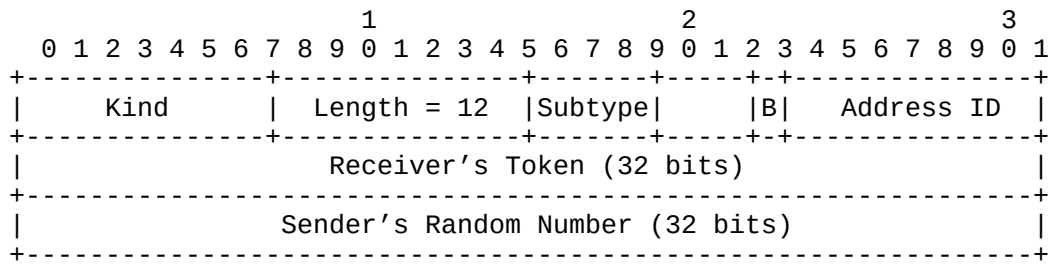


Figure 4: Join Connection (MP_JOIN) Option (for Initial SYN)

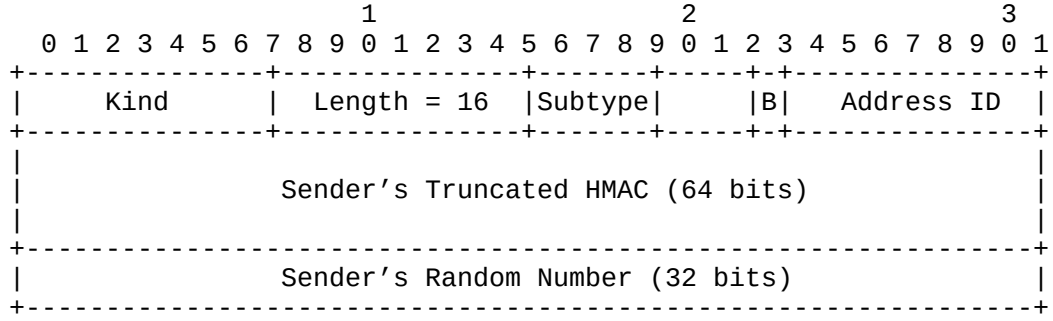


Figure 5: Join Connection (MP_JOIN) Option (for Responding SYN/ACK)

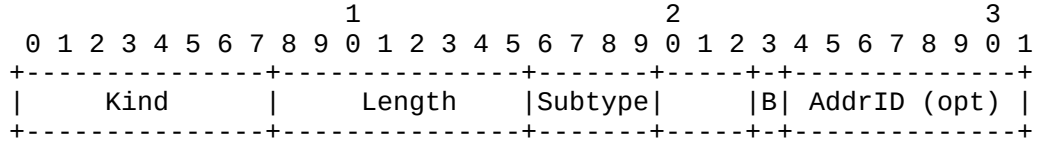


Figure 6: Change Subflow Priority (MP_PRIO) Option

Both solutions rely on the principle that a subflow can be set in backup mode only when being already established or during the subflow setup. On mobile devices, backup subflows consume radio ressources when they are established. This could unnecessarily consume both energy on the mobile device [ATC14] and radio ressources in the network for subflows that do not carry any data. Measurements on smartphones [PAM2016] indicate that many subflows do not carry any data but still consume resources for the SYN, RST and FIN packets.

To allow hosts using Multipath TCP to save ressources, we propose to add the "B" "Backup" Flag in the ADD_ADDR option as shown in Figure 7. This would allow an host to save ressources by being aware

of the remote backup addresses that could be used if all the non-backup subflows fail without having to establish a subflow, achieving a break-before-make scheme.

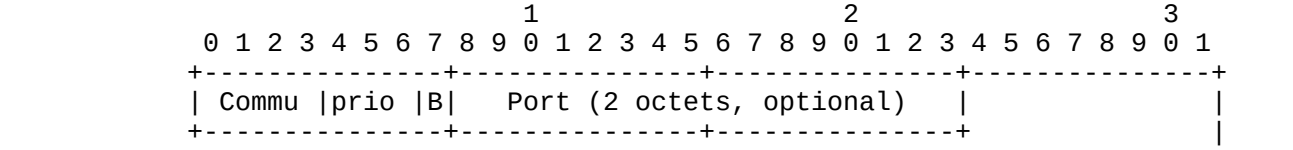


Figure 7: The part of the proposed Multipath TCP new ADD_ADDR option format with the Backup flag

2.3. Priorities

The backup mode defined in [RFC6824] only supports an "all-or-nothing" mode in the usage of the subflows, where an host might just prefer to use certain subflow over others.

To allow an host to inform the receving host about its preference in terms of subflow usage, we propose to modify the ADD_ADDR option by adding 3 "priority" bits as shown in Figure 8.

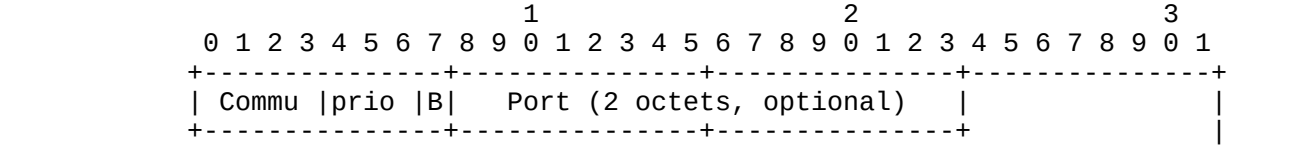


Figure 8: The part of the proposed Multipath TCP new ADD_ADDR option format with the priority bits

This host MAY use this priority to determine when to establish a subflow towards this address. The priority field MUST be interpreted as an unsigned integer value with the highest numerical value being the most preferred one.

To allow the priority of an already established subflow to be modified, we propose to modify the MP_PRIO option by adding the 3 priority bits next to the "B" flag has shown in Figure 9.

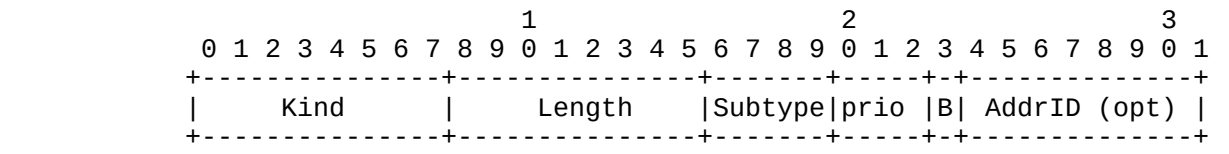


Figure 9: Change Subflow Priority (MP_PRIO) Option with 3 priority bits added

To allow the hosts to advertise a per-subflow priority during the three-way-handshake we modify the MP_JOIN option by adding the 3 priority bits as shown in Figure 10 and Figure 11,

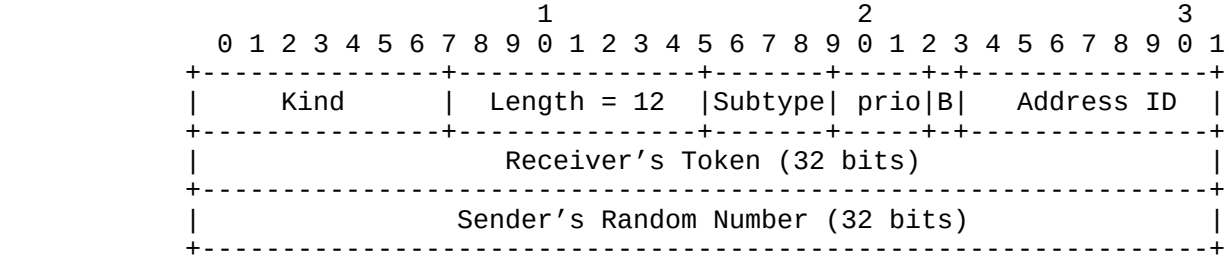


Figure 10: Join Connection (MP_JOIN) Option (for Initial SYN) with the 3 priority bits

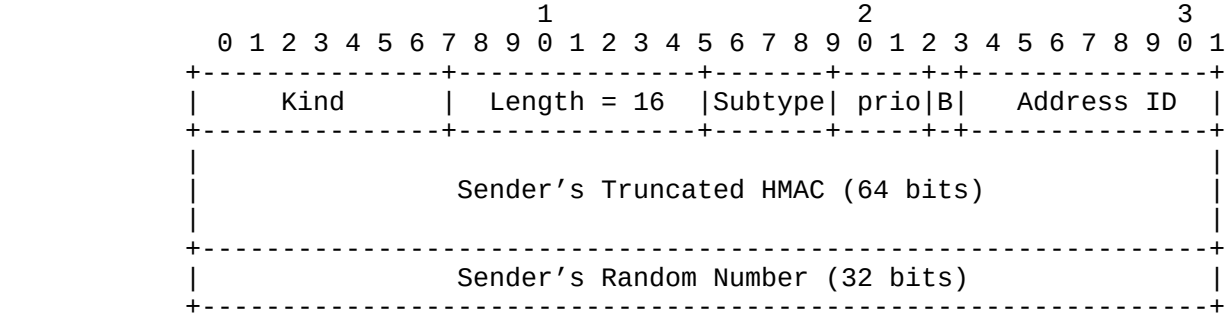


Figure 11: Join Connection (MP_JOIN) Option (for Responding SYN/ACK) with the 3 priority bits

The priority bits included in the MP_JOIN specify indicate the priority associated to this subflow. A host MAY use this information when scheduling packets over this particular subflow.

2.4. Path diversity

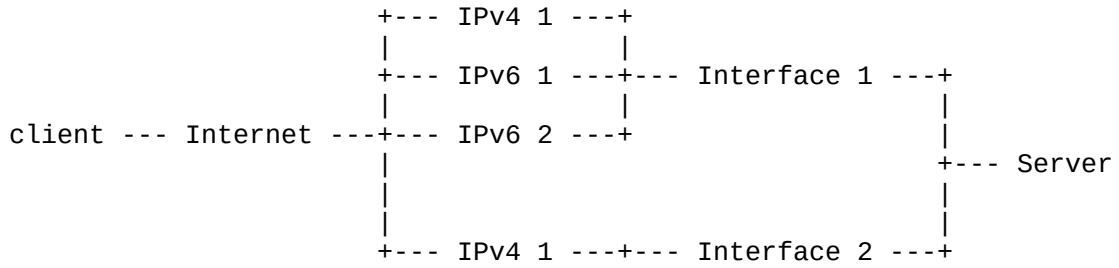


Figure 12: A dual stack server with multiple IP addresses attached to the same interface.

As shown in Figure 12 a host might have several IP addresses assigned to a single interface. Some clients would like to be able to create subflows over disjoint paths to maximise the diversity of the subflows. With the current ADD_ADDR option, the host receiving several ADD_ADDR has no way of knowing the diversity between these path. In the case of Figure 12 it could end up establishing 4 subflows where 2 could be sufficient to maximise diversity.

To allow a host to inform the receiving host about the diversity of several addresses we propose to modify the ADD_ADDR to include 4 bits describing a "Community" associated to this address. The community values are an opaque field and it is expected that two addresses having the same community share some resources.

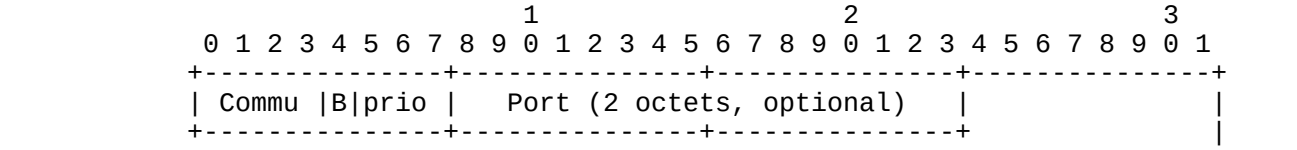


Figure 13: The part of the proposed Multipath TCP new ADD_ADDR option format with the Community bits

With the community bits, a dual-stack host could elect to regroup all the addresses attached to a single interface under the same community, allowing the receiving host to decide on which advertised addresses it wants to establish new subflows.

2.5. Load balancing

Many large web sites are served by servers that are behind a load balancer. The load balancer receives the connection establishment attempts and forwards them to the actual servers that serve the requests. One issue for the end-to-end deployment of Multipath TCP is it ability to be used on load-balancers. Different types of load balancers are possible. We consider a simple but important load balancer than does not maintain any per-flow state. This load balancer is illustrated in illustrated in Figure 14. A stateless load balancer can be implemented by hashing the five tuple (IP addresses and port numbers) of each incoming packet and forward them to one of the servers based on the hash value computed. With TCP, this load balancer ensures that all the packets that belong to one TCP connection are sent to the same server.

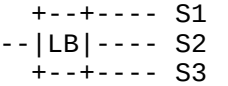


Figure 14: Stateless load balancer

With Multipath TCP, this approach cannot be used anymore when subflows are created by the clients. Such subflows can use any five tuple and thus packets belonging to them will be forwarded over any server, not necessarily the one that was selected by the hashing function for the initial subflow.

To allow Multipath TCP to work for hosts being hosted behind unmodified layer 4 load balancers, we propose to use the unused "B" flag in the MP_CAPABLE option sent (shown in Figure 15 in the SYN+ACK. This flag would allow a host behind a layer 4 load balancer to inform the other host that this address MUST NOT be used to create additional subflows.

A host receiving an MP_CAPABLE with the "B" set to 1 MUST NOT try to establish a subflow to the address used in the MP_CAPABLE. This bit can also be used in the MP_CAPABLE option sent in the SYN by a client that resides behind a NAT or firewall or does not accept server-initiated subflows.

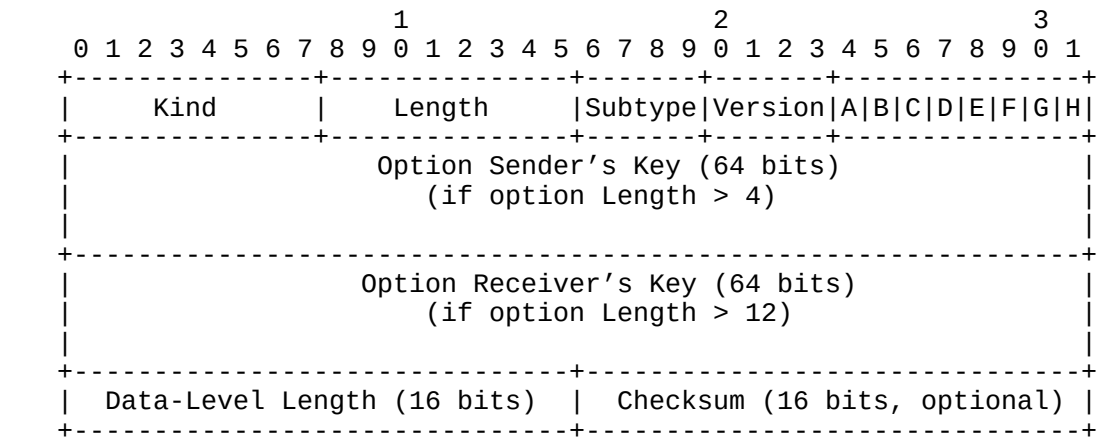


Figure 15: Multipath Capable (MP_CAPABLE) Option

This bit can be used by the servers behind a stateless load balancers. Each of these servers has a different IP address than the address of the load balancer. The servers set the "B" flag in the MP_CAPABLE option that they return and advertise their own address by using the ADD_ADDR option. Upon reception of this option, the clients can create the additional subflows towards these addresses. Compared with current stateless load balancers, an advantage of this approach is that the packets belonging to the additional subflows do not need to pass through the load balancer.

3. IANA considerations

This document proposes some modifications to the Multipath TCP options defined in [RFC6824]. These modifications do not require any specific action from IANA.

4. Security considerations

The security considerations defined for Multipath TCP in [RFC6182] and [RFC7430] are applicable.

The "E" flag, community and priority values in the ADD_ADDR option do not change the security considerations for the handling of this option. Since the ADD_ADDR option is protected by an HMAC, an off-path attacker cannot inject such an option in an existing Multipath TCP connection.

The "priority" field of the MP_PRIO option is not protected by a HMAC. It could be useful to consider the utilisation of an HMAC to protect this option like the ADD_ADDR option.

The "B" flag of the MP_CAPABLE option does not change the security considerations of this option. If an attacker that resides on a path sets this bit, it could prevent the establishment of subflows. However, Multipath TCP does not protect against an attacker that resides on the path of the initial subflow and can modify the SYN/ SYN+ACK packets.

5. Conclusion

In this document, we have discussed several issues with the advertisement of addresses with the address advertisement in Multipath TCP. We have proposed several modifications to the protocol to address these issues.

6. References

6.1. Normative References

[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>.

6.2. Informative References

[ATC14] Yeon-sup Lim, ., Yung-Chih Chen, ., Nahum, Erich., Don Towsley, ., and . Richard Gibbens, "How green is multipath TCP for mobile devices?", AllThingsCellular14 , 2014.

[Cellnet12] Paasch, C., Detal, G., Duchene, F., Raiciu, C., and O. Bonaventure, "Exploring Mobile/WiFi Handover with Multipath TCP", ACM SIGCOMM workshop on Cellular Networks (Cellnet12) , 2012, <http://inl.info.ucl.ac.be/publications/exploring-mobilewifi-handover-multipath-tcp>.

[I-D.ietf-mptcp-rfc6824bis] Ford, A., Raiciu, C., Handley, M., Bonaventure, O., and C. Paasch, "TCP Extensions for Multipath Operation with Multiple Addresses", draft-ietf-mptcp-rfc6824bis-06 (work in progress), July 2016.

[PAM2016] Quentin De Coninck, ., Matthieu Baerts, ., Benjamin Hesmans, ., and O. Bonaventure, "A First Analysis of Multipath TCP on Smartphones", 17th International Passive and Active Measurements Conference , April 2016, <http://inl.info.ucl.ac.be/publications/first-analysis-multipath-tcp-smartphones>.

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

[RFC1323] Jacobson, V., Braden, R., and D. Borman, "TCP Extensions for High Performance", RFC 1323, DOI 10.17487/RFC1323, May 1992, <http://www.rfc-editor.org/info/rfc1323>.

[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, <http://www.rfc-editor.org/info/rfc6182>.

[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, <http://www.rfc-editor.org/info/rfc7430>.

Authors' Addresses

Fabien Duchene
UCLouvain

Email: fabien.duchene@uclouvain.be

Olivier Bonaventure
UCLouvain

Email: Olivier.Bonaventure@uclouvain.be

MPTCP Working Group
Internet-Draft
Intended status: Informational
Expires: January 7, 2017

B. Hesmans
O. Bonaventure
UCLouvain

July 06, 2016

A socket API to control Multipath TCP
draft-hesmans-mptcp-socket-00

Abstract

This document proposes an enhanced socket API to allow applications to control the operation of a Multipath TCP stack.

Status of This Memo

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

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

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

This Internet-Draft will expire on January 7, 2017.

Table of Contents

1. Introduction

Multipath TCP [RFC6824] was designed as an incrementally deployable [RFC6182] extension to TCP [RFC0793]. One of its design objectives was to remain backward compatible with the traditional socket API to enable applications to benefit from Multipath TCP without requiring any modification. This solution has been adopted by the Multipath TCP implementation in the Linux kernel [MultipathTCP-Linux]. In this implementation, once Multipath TCP has been enabled, all TCP applications automatically use it. It is possible to turn Multipath TCP off on a per socket basis, but this is rarely used. The Multipath TCP stack contains a module, called the path manager, that controls the utilisation of the different paths. Three path managers have been implemented :

- the "full mesh" path manager, which is the default one, tries to create subflows in full mesh among all the client addresses and all addresses advertised by the server. All subflows are created by the client because the server assumes that the client is often behind a NAT or firewall

- the "ndiffports" path manager was designed for single-homed hosts. It creates n parallel subflows between the client and the server. It has been defined notably for datacenters [SIGCOMM11]

- the "user space" path manager [CONEXT15] uses Netlink to expose events to specific applications and enables them to control the operation of the underlying MPTCP stack.

However, discussions with users of the Multipath TCP implementation in the Linux kernel indicate that they would often want a finer control on the underlying stack and more precisely on the utilisation of the different subflows. Smartphone applications are a typical example. Measurements indicate that with the default path manager, there are many subflows that are created without being used [PAM2016] [COMMAG2016]. This increases energy consumption and could be avoided on Multipath-TCP aware applications.

The Multipath TCP implementation used in Apple smartphones, tablets and laptops [Apple-MPTCP] took a different approach. This MPTCP stack is not exposed by default to the applications. To use MPTCP, they need to use a specific address family and special system calls [ANRW2016].

Using a new address family and new system calls is a major modification and application developers may not agree to maintain different versions of their applications that run above regular TCP and Multipath TCP. In this document, we propose a simple but powerful API that relies only on socket options and the existing system calls to interact with the MPTCP stack. Application developers are already used to manipulate socket options and could thus easily extend their applications to better utilize the underlying MPTCP stack when available. This approach is similar to the API outlined in [RFC6897], but to our knowledge, this API has never been implemented. We also note that during the last decade the socket API exposed by SCTP evolved to use more socket options [RFC6458].

This document is organised as follows. We first describe the basic operation of our enhanced API in section Section 2. We then show in section Section 3 how the "getsockopt" and "setsockopt" system calls can be used to control the underlying Multipath TCP stack. We focus on basic operations like retrieving the list of subflows that compose a Multipath TCP connection, establishing a new subflow or terminating an existing subflow in this first version of the document. We will address in the next revision of this document more advanced topics such as non-blocking I/O and the utilisation of the "recvmsg" and "sendmsg" system calls.

2. Basic operation

In this section, we briefly describe the basic utilisation of the enhanced socket API for Multipath TCP. As an illustration, we consider a dual-homed smartphone having a WiFi and a cellular interface that interacts with a single homed server.

We assume for simplicity in this example that the server is passive. It creates a listening socket and accepts incoming connections through the following system calls :

- "socket()"
- "bind()"
- "listen()"

Then data can be sent (resp. received) with the "send()" (resp. "recv()") system calls and the connection can be terminated by using the "close()" or "shutdown()" system calls.

On the client side, the following system calls are used to create a Multipath TCP connection :

- "socket()"
- "connect()"

The "connect()" system call succeeds once the initial subflow of the Multipath TCP connection has been established. We assume here that Multipath TCP has been negotiated successfully. The client can then send and receive data by using the "send()" and "recv()" system calls.

The enhanced socket API enables the client (and also the server since the protocol is symmetrical, but we ignore this in this section) to control the utilisation of the different subflows. This control is performed by setting and retrieving socket options through the "setsockopt()" and "getsockopt()" system calls. Four main socket options are defined to control the subflows used by the underlying Multipath TCP connection :

- "MPTCP_GET_SUB_IDS" can only be used by "getsockopt()". It is used to retrieve the current list of the subflows that compose the underlying Multipath TCP connection. In this list, each one identifier is associated with each subflow.
- "MPTCP_GET_SUB_TUPLE". This socket option is equivalent to the "getpeername()" system call with regular TCP, but on a per subflow basis. When used with "getsockopt()", it allows to retrieve the IP addresses and ports of the two endpoints of a particular subflow.
- "MPTCP_OPEN_SUB_TUPLE". This socket option is the equivalent to the "connect()" system call, but it operates on subflows. It

allows to attempt to establish a new subflow by specifying its (remote and optionally local) endpoints.

- "MPTCP_CLOSE_SUB_ID". This socket option allows to close a specific subflow.

As an example, consider a smartphone application that creates a Multipath TCP connection. This connection is established by using the "connect()" system call. The MPTCP stack selects the outgoing interface based on its routing table. Let us assume that the initial subflow is established over the cellular interface. This is the only subflow used for this connection at this time. To perform a handover, the smartphone application would use "MPTCP_OPEN_SUB_TUPLE" to create a new subflow over the WiFi interface. It can then use "MPTCP_GET_SUB_TUPLE" to retrieve the local and remote addresses of this subflow. Now that the WiFi subflow is active, the application can use "MPTCP_CLOSE_SUB_ID" to close the cellular subflow.

3. Multipath TCP Socket API

From an application viewpoint, the interaction with the underlying stack is awlays performed through a single socket. This unique socket is used even if a Multipath TCP stack is used and many subflows have been established. This single socket abstraction is important because the applications exchange data through a bytestream with both TCP and Multipath TCP. We preserve this abstraction in the proposed enhanced socket API but expose some details of the underlying MPTCP stack to the application.

For all the socket options presented bellow, we assume that the underlying Multipath TCP connection is still a Multipath TCP connection. Otherwise (e.g. after a fallback), they return an error and set errno to "EOPNOTSUPP" is returned.

3.1. Subflow list

The first important information that a stack can expose are the different subflows that are combined within a given Multipath TCP connection. For this, we need a data structure that represents the different subflows that compose a connection. The "mptcp_sub_ids" structure shown in figure Figure 1 contains an array with the status of the different subflows that compose a given connection. The actual size of the array depends on the number of subflows and is defined with the "sub_count" field. The "mptcp_sub_status" structure reflects the status of each subflow. A subflow is identified by its "id". In addition to the "id" of the subflow, the "mptcp_sub_status" structure contains one flag : the "low_prio" flag. It is set to 1 when the subflow is defined as a back-up subflow. Other flags could be exposed through this structure in the future.

```
struct mptcp_sub_status {
    __u8     id;
    __u16    low_prio:1;
};

struct mptcp_sub_ids {
    __u8             sub_count;
    struct mptcp_sub_status sub_status[];
};
```

Figure 1: The mptcp_sub_ids and mptcp_sub_status structures

This structure is used by the "MPTCP_GET_SUB_IDS" socket option. More precisely, the "getsockopt", when used with the "MPTCP_GET_SUB_IDS" socket option can retrieve the "mptcp_sub_ids" of the underlying Multipath TCP connection. This call may return an empty array if the connection does not contain any subflow. This can happen with Multipath TCP when the last subflow composing the connection has been terminated abruptly.

The "id" that is returned in the "mptcp_sub_ids" structure is important because it identifies the subflow and is used as an identifier by the other socket options.

The call may return the error "EINVAL" if the buffer passed by the application is too small to copy the array of subflow status.

A simple example of its utilisation is presented in figure Figure 2.

```
int i;
unsigned int optlen;
struct mptcp_sub_ids *ids;

optlen = 42;
```

```
ids = malloc(optlen);

getsockopt(sockfd, IPPROTO_TCP, MPTCP_GET_SUB_IDS, ids, &optlen);

for(i = 0; i < ids->sub_count; i++){
    printf("Subflow id : %i\n",  ids->sub_status[i].id);
}
```

Figure 2: Sample code for the utilisation of MPTCP_GET_SUB_IDS

3.2. Open subflow

Another important part of the API is to enable an application to open new subflows. This is possible through the "MPTCP_OPEN_SUB_TUPLE" socket option. This option uses the "mptcp_sub_tuple" structure shown in figure Figure 3 to pass the priority, local and remote endpoints of the new subflow.

```
struct mptcp_sub_tuple {
    __u8    id;
    __u8    prio;
    __u8    addrs[0];
};
```

Figure 3: The mptcp_sub_tuple structure

The "id" field is an output. This is the "id" of the created subflow. The "prio" field indicates if the new subflow should be considered as back-up or not. The "addrs" must be a pair array of size two. The first address must be the address of the source and the second address must be the address of the destination. The actual structure passed must be either "sockaddr_in"or "sockaddr_in6", but the two elements of the array must be of the same type. The struct "sockaddr" can be used to determine which one is actually passed.

The caller can also set the source address to be either "INADDR_ANY" for IPv4 or "in6addr_any" for IPv6. In this case, the kernel chooses the source address to be used for the new subflow.

Errors returned by either "bind()" or "connect()" are returned if an error occurred during the process.

An example is provided in figure Figure 4.

```
unsigned int optlen;
struct mptcp_sub_tuple *sub_tuple;
struct sockaddr_in *addr;
```

```
int error;

optlen = sizeof(struct mptcp_sub_tuple) +
         2 * sizeof(struct sockaddr_in);
sub_tuple = malloc(optlen);

sub_tuple->id = 0;
sub_tuple->prio = 0;

addr = (struct sockaddr_in*) &sub_tuple->addrs[0];

addr->sin_family = AF_INET;
addr->sin_port = htons(12345);
inet_pton(AF_INET, "10.0.0.1", &addr->sin_addr);

addr++;

addr->sin_family = AF_INET;
addr->sin_port = htons(1234);
inet_pton(AF_INET, "10.1.0.1", &addr->sin_addr);

error =  getsockopt(sockfd, IPPROTO_TCP, MPTCP_OPEN_SUB_TUPLE,
                    sub_tuple, &optlen);
```

Figure 4: Sample code to establish an additional subflow

3.3. Close subflow

To close a subflow, the socket option "MPTCP_CLOSE_SUBFLOW" is used. This option used the "mptcp_close_sub_id" structure defined in figure Figure 5.

```
struct mptcp_close_sub_id {
    __u8    id;
    int     how;
};
```

Figure 5: The mptcp_close_sub_id structure

In the above structure, "id" is the identifier of the subflow that needs to be closed. If the "id" is invalid, "EINVAL" is returned.

The "how" field is used to define how to subflow should be terminated. It recognises the same set of constant that are used by "shutdown()". In addition to this set, "RST" can be used to indicates that the subflow should be terminated by sending an "RST".

3.4. Get subflow tuple

An application may also be interested by the addresses and ports that are used by a given subflow. To retrieve this information, the socket option "MPTCP_GET_SUB_TUPLE" is used in combination with the "mptcp_sub_tuple" structure shown in figure Figure 6.

```
struct mptcp_sub_tuple {
    __u8    id;
    __u8    addrs[0];
};
```

Figure 6: The mptcp_sub_tuple structure

This is the same structure as the one used to open a subflow but in this context, "id" is the input and "addrs" is the output.

A sample code is provided in figure Figure 7.

```
unsigned int optlen;
struct mptcp_sub_tuple *sub_tuple;

optlen = 100;

sub_tuple = malloc(optlen);

sub_tuple->id = sub_id;
getsockopt(sockfd, IPPROTO_TCP, MPTCP_GET_SUB_TUPLE, sub_tuple,
           &optlen);

sin = (struct sockaddr_in*) &sub_tuple->addrs[0];

printf("\tip src : %s src port : %hu\n", inet_ntoa(sin->sin_addr),
                                         ntohs(sin->sin_port));

sin++;

printf("\tip dst : %s dst port : %hu\n", inet_ntoa(sin->sin_addr),
                                         ntohs(sin->sin_port));
```

Figure 7: Sample code using the MPTCP_GET_SUB_TUPLE option

3.5. Subflow socket option

TCP/IP implementations support different socket options. Some of them can be applied to the TCP layer while others can be applied to the IP layer. To be able to issue a socket option on a specific subflow, we define the "MPTCP_SUB_GETSOCKOPT" and

"MPTCP_SUB_SETSOCKOPT" options. These two socket options use respectively the structures presented in figure Figure 8.

```
struct mptcp_sub_getsockopt {
    __u8        id;
    int        level;
    int        optname;
    char __user    *optval;
    unsigned int __user    *optlen;
};

struct mptcp_sub_setsockopt {
    __u8        id;
    int        level;
    int        optname;
    char __user    *optval;
    unsigned int    optlen;
};
```

Figure 8: Structures used by the ‘‘MPTCP_SUB_GETSOCKOPT‘‘ and ‘‘MPTCP_SUB_SETSOCKOPT‘‘ options

In the two structures "id" indicates to which subflow the socket option should be redirected. The end of each structure contains the information needed to perform the socket option call on the subflow.

Figure Figure 9 illustrates how the IP_TSO socket option can be applied on a particular subflow.

```
unsigned int optlen, sub_optlen;
struct mptcp_sub_setsockopt sub_sso;
int val = 12;

optlen = sizeof(struct mptcp_sub_setsockopt);
sub_optlen = sizeof(int);
sub_sso.id = sub_id;
sub_sso.level = IPPROTO_IP;
sub_sso.optname = IP_TOS;
sub_sso.optlen = sub_optlen;
sub_sso.optval = (char *) &val;

setsockopt(sockfd, IPPROTO_TCP, MPTCP_SUB_SETSOCKOPT, &sub_sso,
           optlen);
```

Figure 9: Example socket option

4. IANA considerations

There are no IANA considerations in this document.

5. Security considerations

TCP and UDP implementations usually reserve port numbers below 1024 for privileged users. On such implementations, Multipath TCP should restrict the ability of the users to create subflows on privileged ports through the "MPTCP_OPEN_SUB_TUPLE".

For similar reasons, the "MPTCP_SUB_SETSOCKOPT" socket option should not enable an unprivileged user to retrieve or modify a socket option on a subflow if he is not allowed to perform such actions on a regular TCP connection.

Applications requiring strong security should implement cryptographic protocols such as TLS [RFC5246] or ssh [RFC4251]. The proposed API enables such application to better control their utilisation of the underlying interfaces by managing the different subflows.

6. Conclusion

In this document, we have documented an enhanced socket API that enables applications to control the creation and the release of subflows by the underlying Multipath TCP stack. We expect that a standardised API supported by different implementations will be an important stop for the deployment of Multipath TCP aware applications on both multihomed hosts such as smartphones as well as on servers. This enhanced API has already been implemented on the Multipath TCP implementation in the Linux kernel. Future versions of this document will address more advanced utilisations of the socket API such as non-blocking I/O and the "sendmsg()" and "recvmsg()" system calls.

7. Acknowledgements

We would like to thank Christoph Paasch, Quentin De Coninck Rao Shoaib for their comments on an early version of this document.

8. References

8.1. Normative References

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

8.2. Informative References

[ANRW2016]
Hesmans, B. and O. Bonaventure, "An enhanced socket API for Multipath TCP", 2016, <https://irtf.org/anrw/2016/anrw16-final16.pdf>.

[Apple-MPTCP]
Apple, Inc, ., "iOS - Multipath TCP Support in iOS 7", n.d., <https://support.apple.com/en-us/HT201373>.

[COMMAG2016]
De Coninck, Q., Baerts, M., Hesmans, B., and O. Bonaventure, "Observing Real Smartphone Applications over Multipath TCP", IEEE Communications Magazine , March 2016, <http://inl.info.ucl.ac.be/publications/observing-real-smartphone-applications-over-multipath-tcp>.

[CONEXT15]
Hesmans, B., Detal, G., Barre, S., Bauduin, R., and O. Bonaventure, "SMAPP - Towards Smart Multipath TCP-enabled APPlications", Proc. Conext 2015, Heidelberg, Germany , December 2015, <http://inl.info.ucl.ac.be/publications/smapp-towards-smart-multipath-tcp-enabled-applications>.

[MultipathTCP-Linux]
Paasch, C., Barre, S., and . et al, "Multipath TCP implementation in the Linux kernel", n.d., <http://www.multipath-tcp.org>.

[PAM2016] De Coninck, Q., Baerts, M., Hesmans, B., and O. Bonaventure, "A First Analysis of Multipath TCP on Smartphones", 17th International Passive and Active Measurements Conference (PAM2016) , March 2016, <http://inl.info.ucl.ac.be/publications/first-analysis-multipath-tcp-smartphones>.

[RFC4251] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Protocol Architecture", RFC 4251, DOI 10.17487/RFC4251, January 2006, <http://www.rfc-editor.org/info/rfc4251>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/

RFC5246, August 2008,
<http://www.rfc-editor.org/info/rfc5246>.

[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, <http://www.rfc-editor.org/info/rfc6182>.

[RFC6458] Stewart, R., Tuexen, M., Poon, K., Lei, P., and V. Yasevich, "Sockets API Extensions for the Stream Control Transmission Protocol (SCTP)", RFC 6458, DOI 10.17487/RFC6458, December 2011, <http://www.rfc-editor.org/info/rfc6458>.

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

[SIGCOMM11] Raiciu, C., Barre, S., Pluntke, C., Greenhalgh, A., Wischik, D., and M. Handley, "Improving datacenter performance and robustness with multipath TCP", Proceedings of the ACM SIGCOMM 2011 conference , 2011, <http://doi.acm.org/10.1145/2018436.2018467>.

Authors' Addresses

Benjamin Hesmans
UCLouvain

Email: Benjamin.Hesmans@uclouvain.be

Olivier Bonaventure
UCLouvain

Email: Olivier.Bonaventure@uclouvain.be

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

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

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

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any

time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on December 10, 2019.

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:

- 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.

- 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.

- 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:

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

- 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].

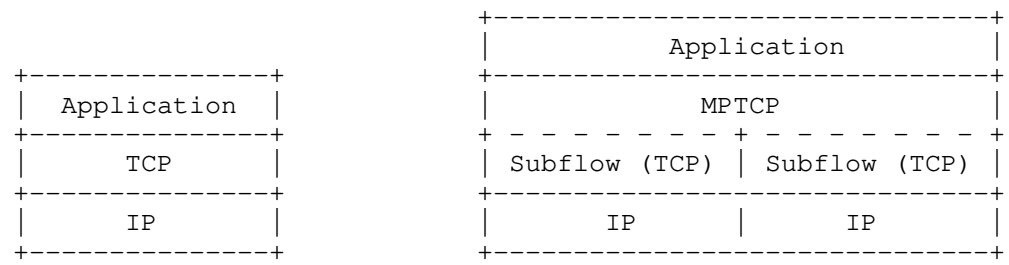


Figure 1: Comparison of Standard TCP and MPTCP Protocol Stacks

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.

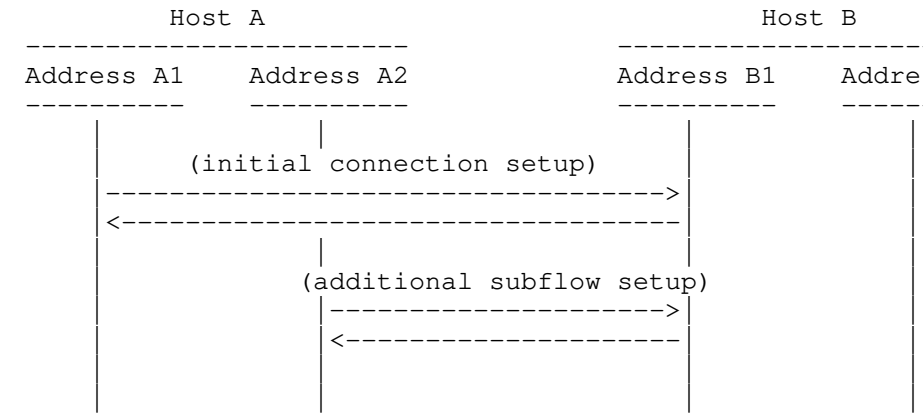


Figure 2: Example MPTCP Usage Scenario

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]
```

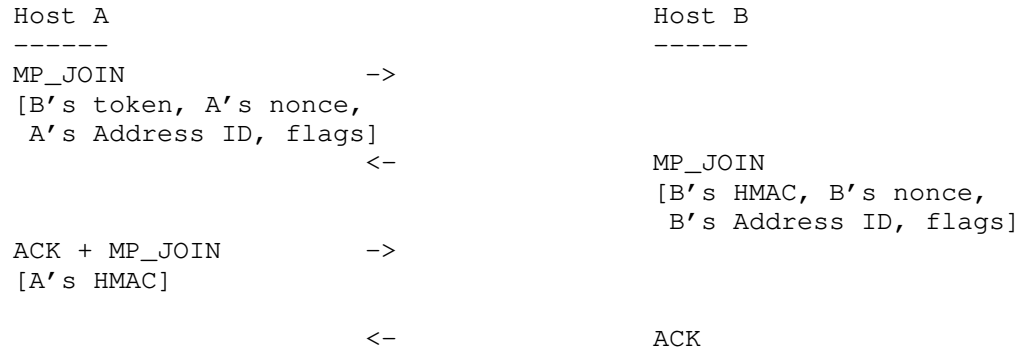
```
Host A                                  Host B (sending first)
------                                  ------
MP_CAPABLE                 ->
[flags]
                           <-           MP_CAPABLE
                                        [B's key, flags]
ACK + MP_CAPABLE           ->
[A's key, B's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



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 subflows] ->

                               <-          [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:

- 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.

- 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.

- 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.

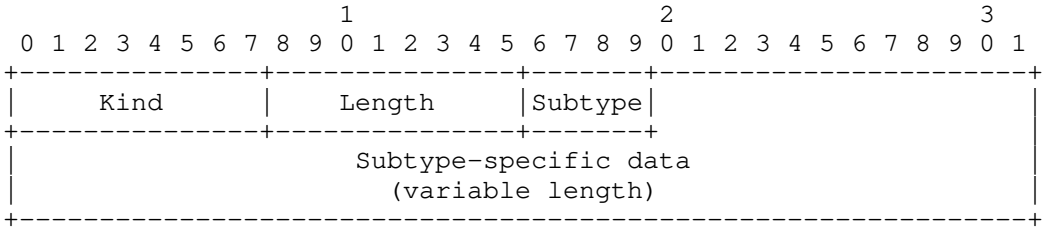


Figure 3: MPTCP Option Format

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.

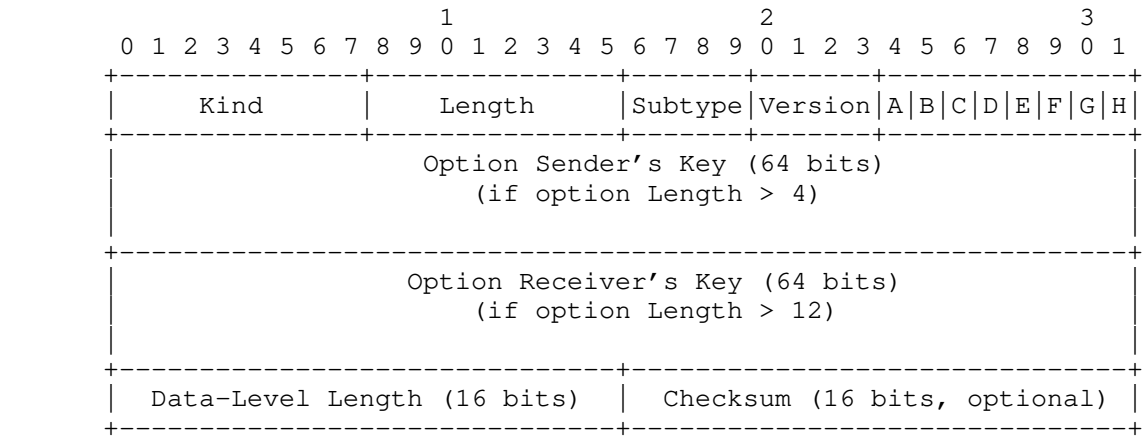


Figure 4: Multipath Capable (MP_CAPABLE) Option

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.

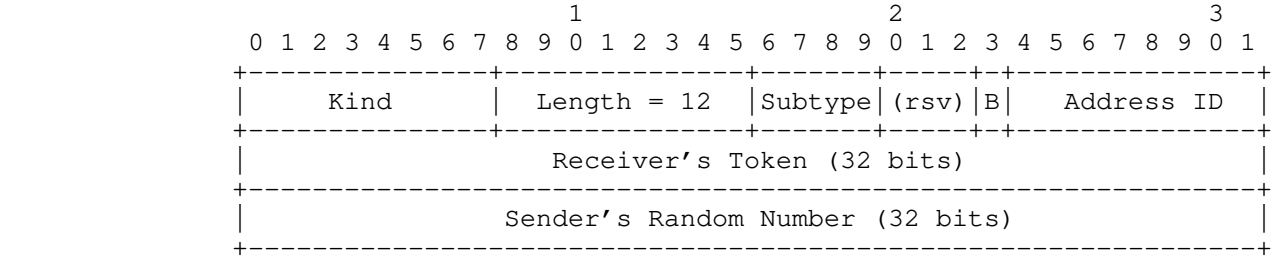


Figure 5: Join Connection (MP_JOIN) Option (for Initial SYN)

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.

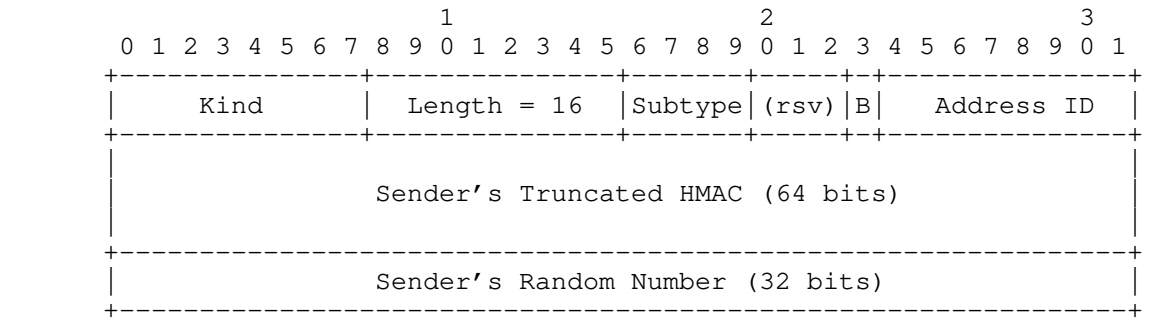


Figure 6: Join Connection (MP_JOIN) Option (for Responding SYN/ACK)

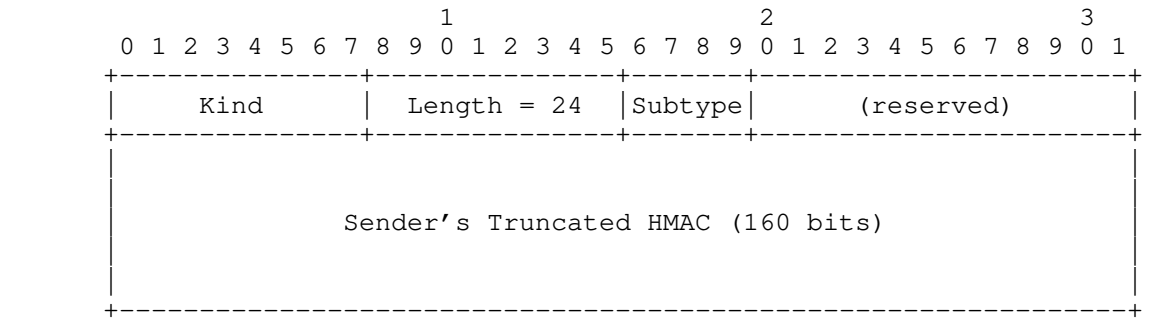
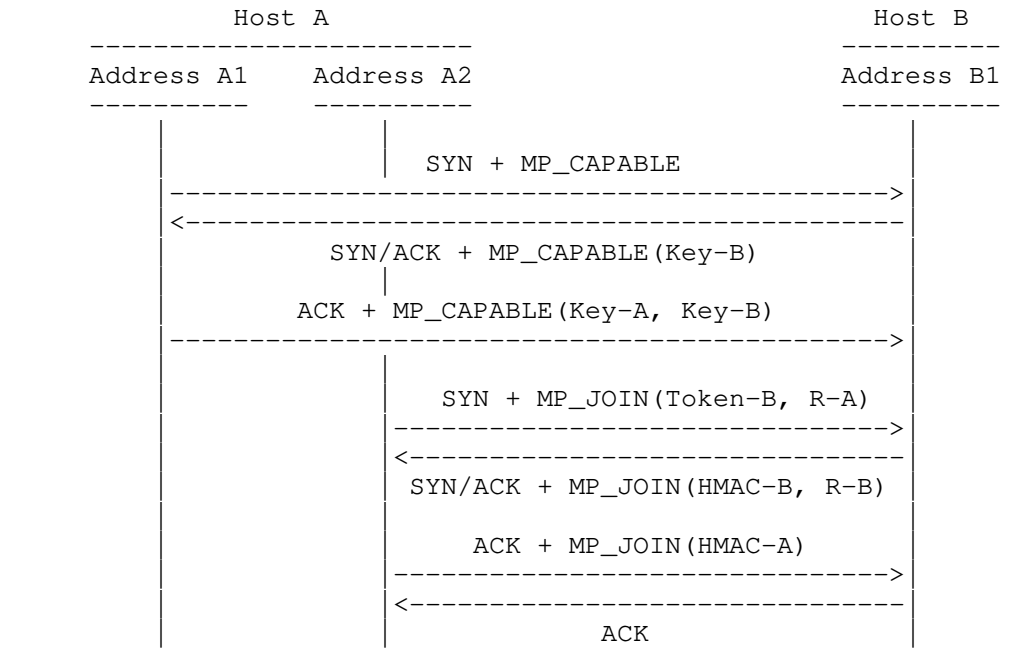


Figure 7: Join Connection (MP_JOIN) Option (for Third ACK)

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



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.

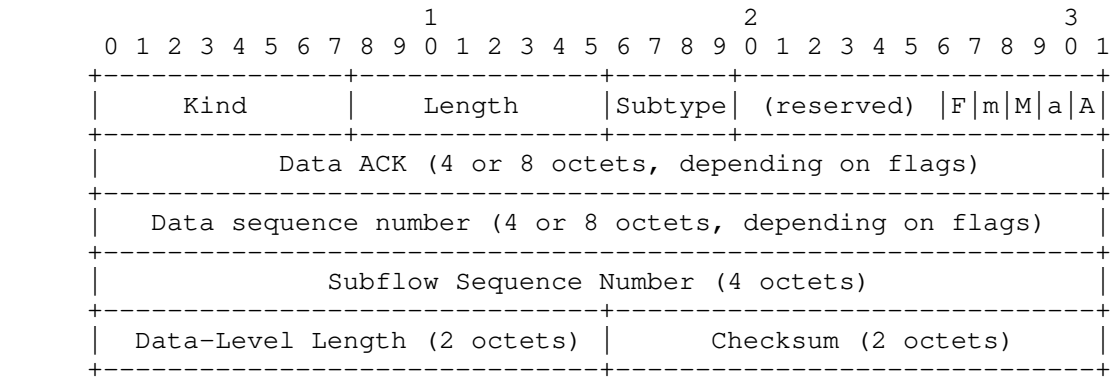


Figure 9: Data Sequence Signal (DSS) Option

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

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

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

The 'F' flag indicates "Data FIN". If present, this means that this mapping covers the final data from the sender. This is the connection-level equivalent to the FIN flag in single-path TCP. A connection is not closed unless there has been a Data FIN exchange, 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.

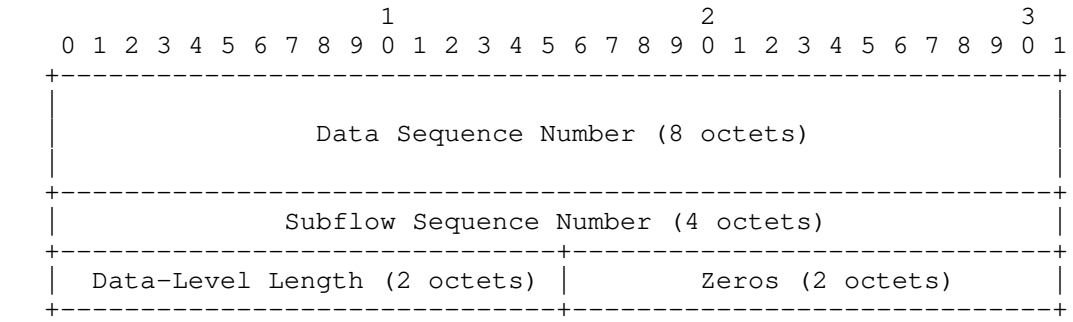


Figure 10: Pseudo-Header for DSS Checksum

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.

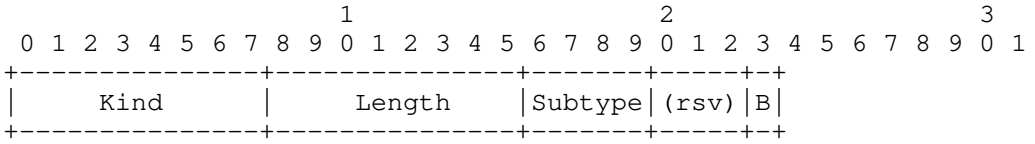


Figure 11: Change Subflow Priority (MP_PRIO) Option

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.

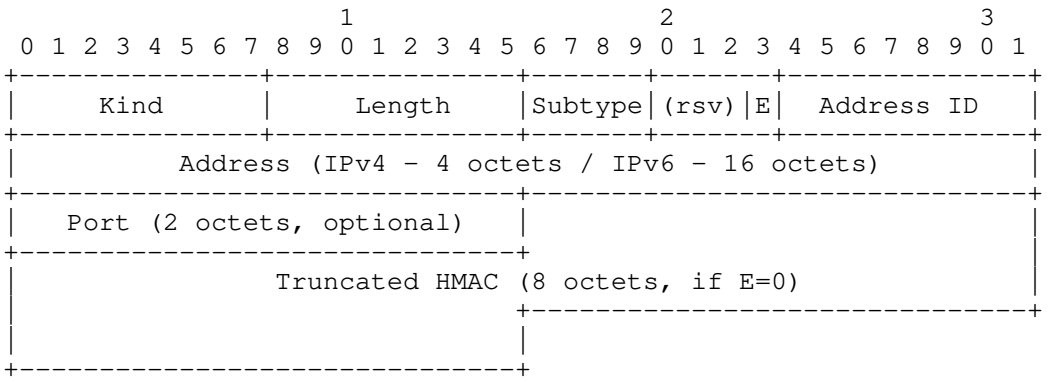


Figure 12: Add Address (ADD_ADDR) Option

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.

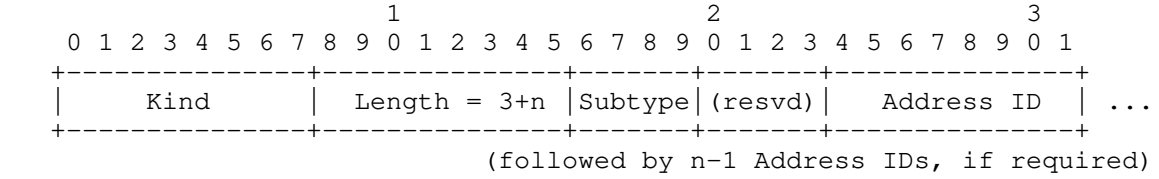


Figure 13: Remove Address (REMOVE_ADDR) Option

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.

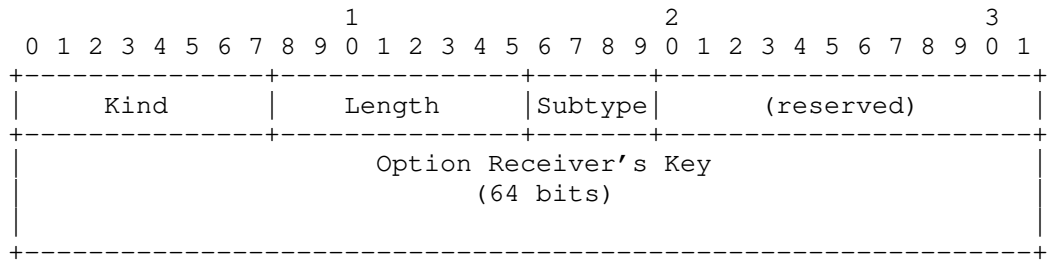


Figure 14: Fast Close (MP_FASTCLOSE) Option

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.

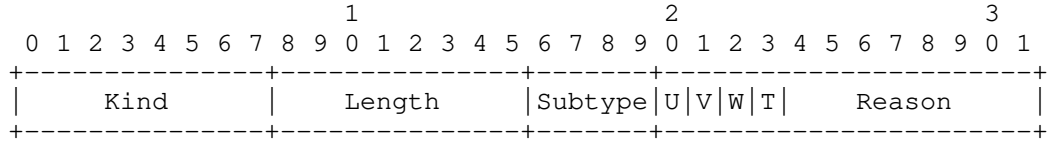


Figure 15: TCP RST Reason (MP_TCPRST) Option

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:

- 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.

- 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.

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

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

- 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.

- 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.

- 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.

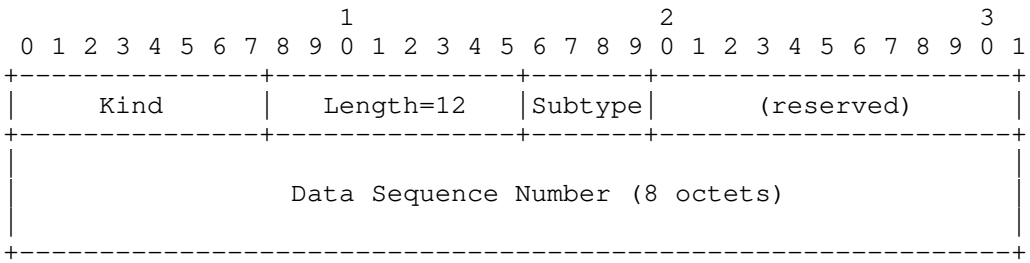


Figure 16: Fallback (MP_FAIL) Option

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:

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

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

- 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.

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

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

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

- o available flags in MP_CAPABLE (Figure 4);

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

- o the version field in MP_CAPABLE (Figure 4);

6. Interactions with Middleboxes

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

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

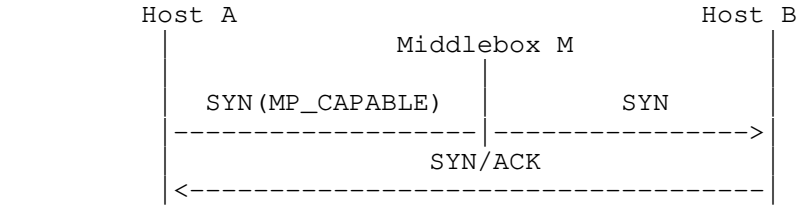
MPTCP uses a single new TCP option "Kind", and all message types are defined by "subtype" values (see Section 8). This should reduce the chances of only some types of MPTCP options being passed, and instead the key differing characteristics are different paths, and the presence of the SYN flag.

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

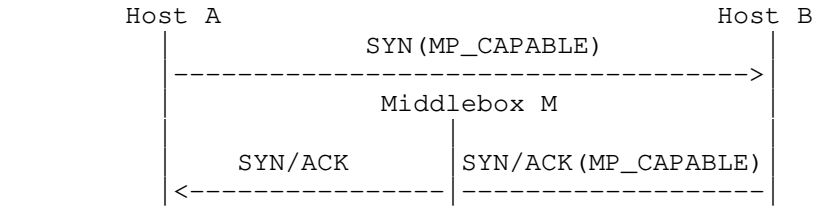
If a middlebox strips options but otherwise passes the packets unchanged, MPTCP will behave safely. If an MP_CAPABLE option is

dropped on either the outgoing or the return path, the initiating host can fall back to regular TCP, as illustrated in Figure 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.



a) MP_CAPABLE option stripped on outgoing path



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.

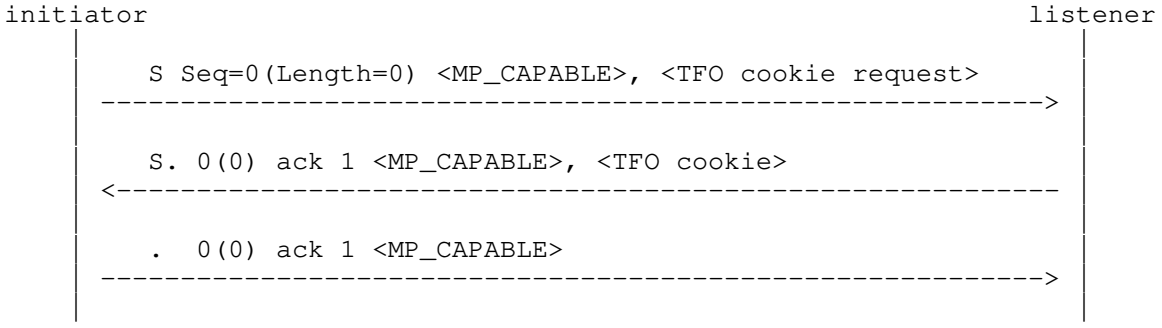


Figure 18: Cookie request - sequence number and length are annotated as Seq(Length) and used hereafter in the figures.

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.

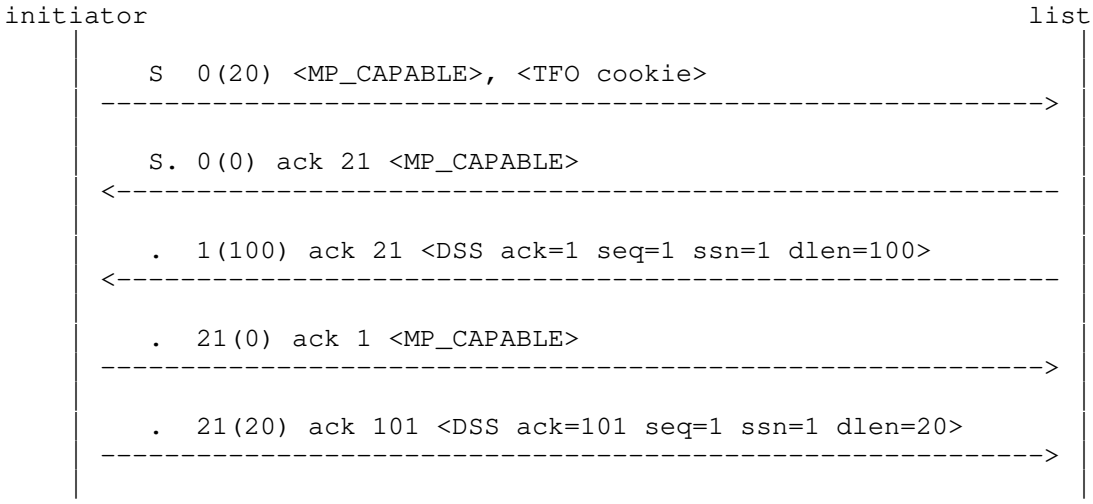


Figure 19: The listener supports TFO

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.

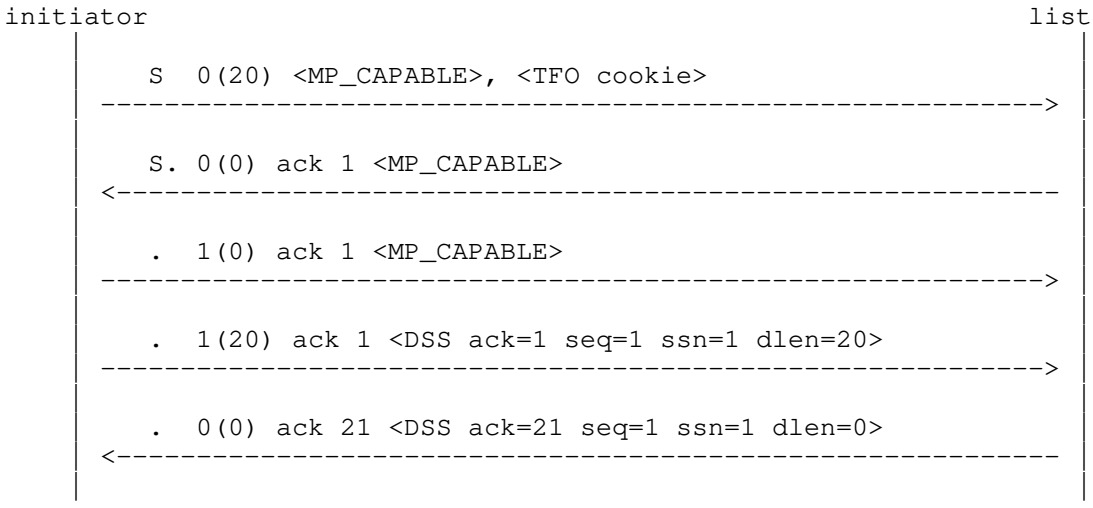


Figure 20: The listener does not support TFO

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.

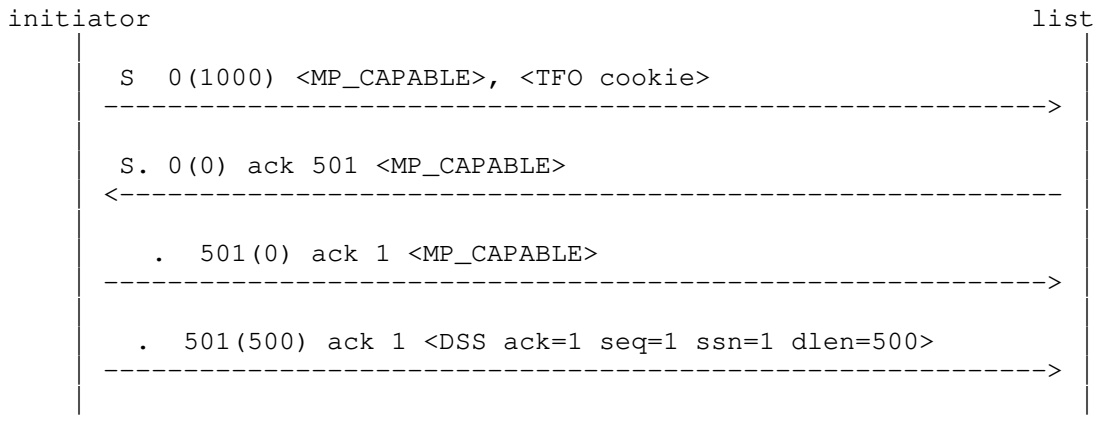


Figure 21: Partial data acknowledgement

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.

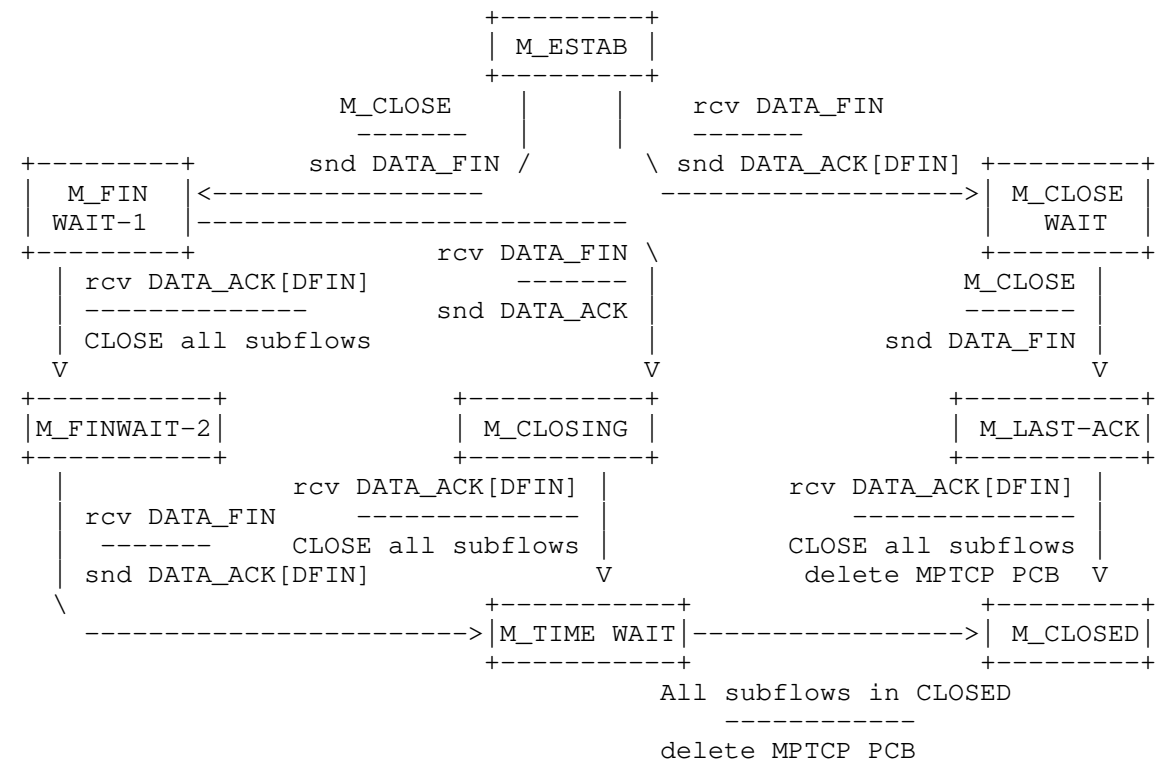


Figure 22: Finite State Machine for Connection Closure

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.

- 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].

- 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 Engineering Task Force
Internet-Draft
Intended status: Experimental
Expires: November 28, 2016

C. Paasch
Apple, Inc.
A. Ford
Pexip

May 27, 2016

Application Layer Authentication for MPTCP
draft-paasch-mptcp-application-authentication-00

Abstract

Multipath TCP (MPTCP), described in [3], is an extension to TCP to provide the ability to simultaneously use multiple paths between hosts.

MPTCP currently specifies a single authentication mechanism, using keys that are initially exchanged in the clear. There are application-layer protocols that may have better information as to the identity of the parties and so is able to better provide keying material that could be used for the authentication of future subflows.

This document specifies "application layer authentication" for Multipath TCP, an alternatively negotiated keying mechanism for MPTCP.

Status of This Memo

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

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

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

This Internet-Draft will expire on November 28, 2016.

Table of Contents

1. Introduction

The MPTCP handshake serves multiple purposes. First, hosts discover their peer's support of MPTCP. Second, each host announces a key that will be tied to this MPTCP session. The key also serves multiple purposes. First, the derivate of the key is being used as a token-identifier for the MPTCP connection. This derivate is a truncated hash of the key. Second, another truncated hash of the key serves as the initial data sequence number. And third, the key itself is used as an authenticator to prove that the host behind the IP-address used to establish new subflows is indeed the one that participated in the handshake of the initial subflow.

In the following we explain the shortcomings of this exchange and how they impact the deployment of MPTCP.

1.1. Key in plaintext

The key-exchange happens during the handshake of the initial subflow. RFC 6824 specifies that this exchange happens in plaintext. As has been noted in RFC 7430, an eavesdropper on the initial handshake is thus able to learn the keys used in this MPTCP session. This allows him to generate the session's tokens and data sequence numbers, enabling him to effectively hijack the MPTCP session by creating a subflow with a different IP-address. The attacker will be able to generate a valid HMAC as he has full knowledge of the keys of this MPTCP session.

To enhance MPTCP's security, it would be beneficial to not reveal MPTCP's keys in plaintext on the wire.

1.2. Token generation

The token is a truncation of the 32 most significant bits of the SHA-1 of the key. The key must be a random number of sufficient entropy to be used as part of the authentication mechanism, and thus a host has no control over the token as it is generating the key for the MPTCP-session. This has some implications on the deployability of MPTCP, outlined hereafter.

1.2.1. Hash collision

Due to the nature of the token-generation, the 32-bit token might collide with another already existing MPTCP session. While a 32-bit token collision should be very rare on client devices, a busy server (with potentially tens of millions of active MPTCP connections) will have a very high probability of a token collision.

Upon such a collision, the server needs to generate a new cryptographically secure 64-bit key, and derive the token through a SHA-1 computation upon which he finally can verify the uniqueness of the token. If a collision happened again, the server has to start anew. This process imposes a computation overhead and complexity upon the server and impacts the scalability compared to regular TCP. Allowing a server to generate a token in such a way that uniqueness can be achieved easily would be beneficial for the scalability and deployment of MPTCP.

1.2.2. Derive information from the token

As the token is a truncated hash of the key, it is entirely of a random nature. As has been shown in [5], this brings several deployment challenges in large server farms. In particular, the

layer-4 load balancers in front of this server farm need to maintain MPTCP-specific state in order to map a token to the server.

The token can be looked at as a route-identifier, as it allows the server to associate the incoming SYN+MP_JOIN with an existing MPTCP-session. However, the random nature of the token does not allow a load balancer in the middle to do the same without having to maintain MPTCP-specific state.

If the token can be generated in such a way that it carries the required routing information in such a way that it can be deciphered by all the trusted parties in the server farm deployment, large-scale deployment of MPTCP would be simplified.

In the following we suggest an alternative handshake that allows MPTCP to increase its security by leveraging an external key-exchange and thus benefit from the security provided by protocols like TLS. As a side-effect of this approach, the token also can be exchanged in a more flexible way, addressing the above identified issues with the token generation.

2. Proposed Technical Changes

2.1. MP_CAPABLE Changes

To resolve the issues identified in the previous section, this proposal separates the key handling for security (i.e. the method for protecting new subflow exchanges) from the token exchange. This means that:

- Key exchange is handled in the application layer
- Meaning can be exchanged in the token, and a custom generation method can be used, as it is decoupled from keying material

This specification allocates the ’G’ bit from the flags of MP_CAPABLE as an alternative security mechanism - "handled by application layer". In this case, the MP_CAPABLE exchange will send and receive tokens rather than keys.

When the ’G’ bit is set to 1, this implies support for this new mechanism, and the MP_CAPABLE exchange will operate as follows. The tokens take the place of the keys in the MP_CAPABLE exchange, but otherwise the exchange remains very similar. This exchange still maintains support for stateless servers. Note that this now means that tokens are 64 bits in length.

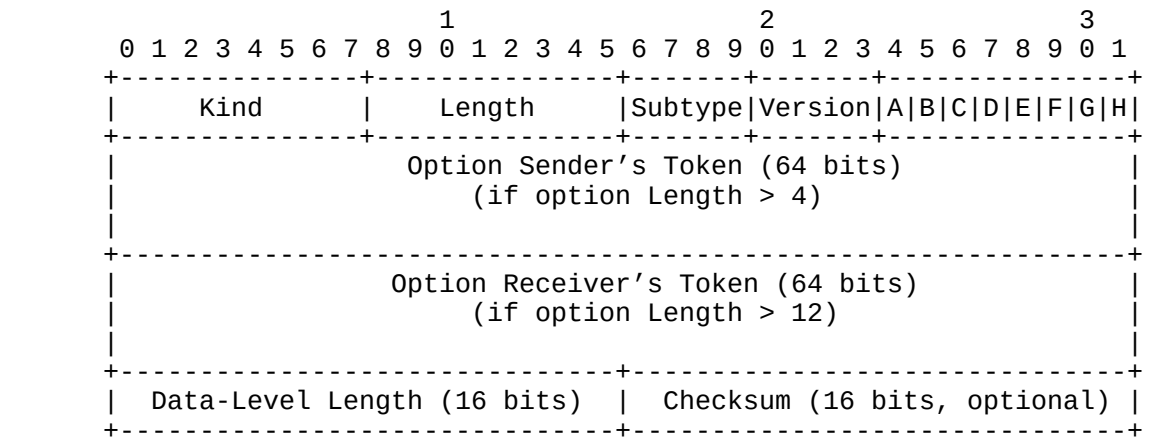


Figure 1: Proposed Multipath Capable (MP_CAPABLE) Option

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.

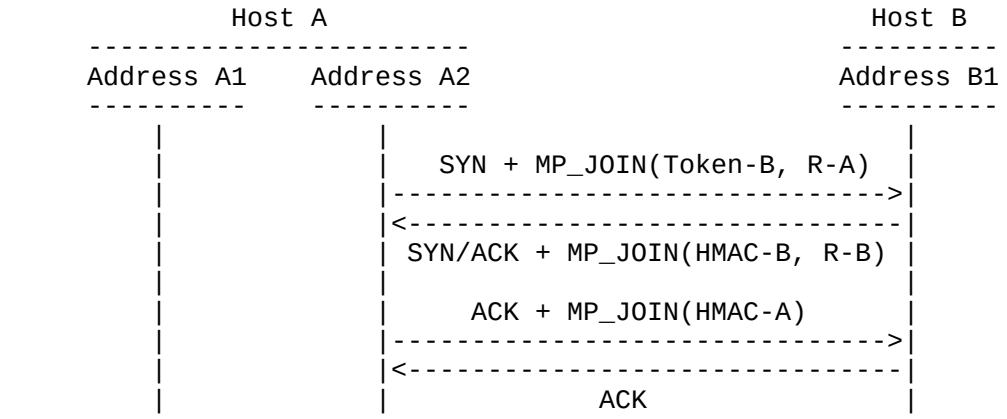
- SYN (A->B): only the first four octets (Length = 4).
- SYN/ACK (B->A): B's token for this connection (Length = 12).
- ACK (no data) (A->B): A's token followed by B's token (Length = 20).
- ACK (with first data) (A->B): A's key followed by B's key followed by Data-Level Length, and optional Checksum (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 1, "sender" and "receiver" refer to the sender or receiver of the TCP packet (which can be either host).

If the sender of the initial SYN supports both SHA-1 (as specified in [3]) and application-layer, it can set both G and H bits to "1". The sender of the SYN/ACK can then make a decision as to which mode to support, and selects only one of those bits in the SYN/ACK.

2.2. MP_JOIN Changes

The MP_JOIN exchange remains almost the same:



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 2: Example Use of MP_JOIN

However, the token presented is now 64 bits. The key used in the HMAC exchange here is provided by the application layer. Otherwise, there are no other changes to the handshake. Note, however, that an MP_JOIN message cannot be sent until the application layer protocol has determined that the key exchange has completed.

Depending on the key-exchange protocol that is in use at the application layer, it may be that the client already knows the key, while the server is not yet aware of it. In that case the server might receive SYN+MP_JOIN with a valid token, but the MPTCP-state on the server has not yet been populated with the key. The server must silently drop in that case the SYN+MP_JOIN. The client will retransmit its SYN+MP_JOIN and eventually the application on the server will have populated the MPTCP-state with the key.

2.3. Data Sequence Number Changes

The Initial Data Sequence Number for each host involved in an MPTCP connection is, by [3], derived from the SHA-1 hash of the key. If application-layer authentication is selected, the IDSN MUST instead be derived from the most-significant 64 bits of the SHA-1 hash of the token.

2.4. MP_FASTCLOSE Changes

MP_FASTCLOSE is the other method that uses the key in [3]. Given there is no knowledge as to a potential key's sensitivity, it can no longer be said that a key should be sent here. Instead, a truncation of the 64 most-significant bits of the SHA-1 hash [4] of the key should be used.

3. Security Considerations

This draft is proposing a mechanism that would allow an application-layer protocol to provide security, rather than relying on a cleartext exchange of the keys. As such, this document itself does not introduce any additional security concerns, but provides a mechanism by which additional security could be added to the MPTCP handshake, depending on the authentication method used at the application layer.

4. IANA Considerations

This document would update the "MPTCP Handshake Algorithms" sub-registry under the "Transmission Control Protocol (TCP) Parameters" registry, based on the flags in MP_CAPABLE, to add the following algorithm:

Flag Bit	Meaning	Reference
G	Application-layer Authentication	This document

Table 1: MPTCP Handshake Algorithms

5. References

5.1. Normative References

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

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

[3] Ford, A., Raiciu, C., Handley, M., Bonaventure, O., and C. Paasch, "TCP Extensions for Multipath Operation with Multiple Addresses", draft-ietf-mptcp-rfc6824bis-05 (work in progress), January 2016.

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

5.2. Informative References

[5] Paasch, C., Greenway, G., and A. Ford, "Multipath TCP behind Layer-4 loadbalancers", draft-paasch-mptcp-loadbalancer-00 (work in progress), September 2015.

Authors' Addresses

Christoph Paasch
Apple, Inc.
Cupertino
US

EMail: cpaasch@apple.com

Alan Ford
Pexip

EMail: alan.ford@gmail.com

Internet Engineering Task Force
Internet-Draft
Intended status: Experimental
Expires: November 28, 2016

C. Paasch
Apple, Inc.
A. Ford
Pexip

May 27, 2016

TLS Authentication for MPTCP
draft-paasch-mptcp-tls-authentication-00

Abstract

Multipath TCP (MPTCP), described in [4], is an extension to TCP to provide the ability to simultaneously use multiple paths between peers.

draft-paasch-mptcp-application-authentication specifies "application layer authentication" for Multipath TCP, an alternatively negotiated keying mechanism for MPTCP. This allows keying material to be sourced from an application layer protocol in order to secure MP_JOIN handshakes.

This document explains how to use the proposed application-layer authentication extension with TLS [6], in order to leverage securely exchanged keys for MPTCP security, whilst simultaneously freeing the MPTCP token to be used as a channel for additional information.

Status of This Memo

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

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

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

This Internet-Draft will expire on November 28, 2016.

Table of Contents

1. Introduction

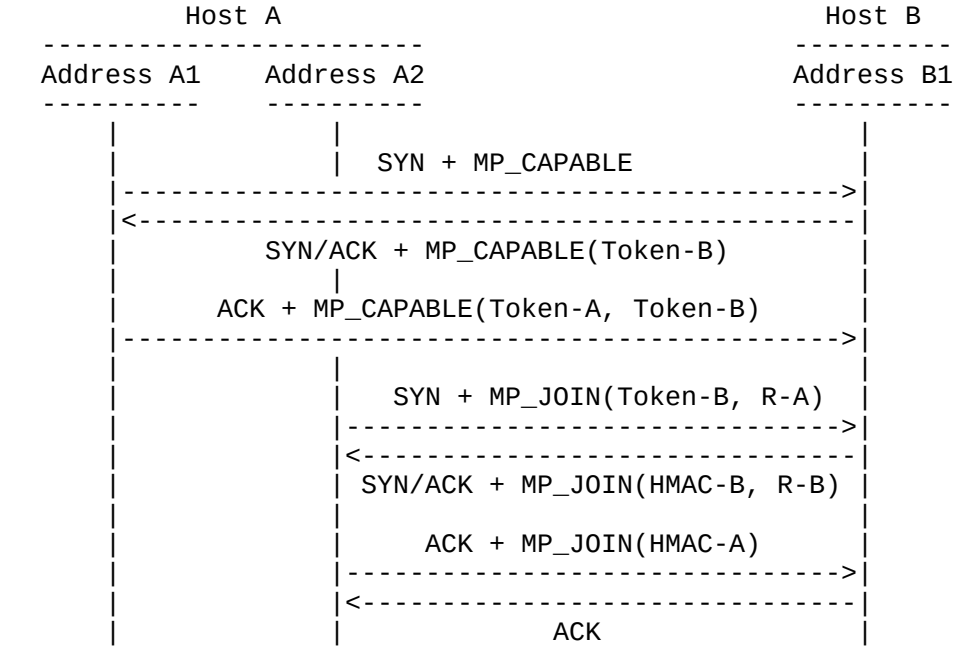
As described in draft-paasch-mptcp-application-authentication, the use of "application-layer authentication" allows the Key used in MPTCP authentication to be provided by the application layer, thus permitting the use of existing secure communication channels for exchanging keying material. Furthermore, this decouples the key from the token and thus allows the token to be used for conveying additional semantics, such as helping front-end proxies route traffic to appropriate back-end servers.

TLS [6] provides a secure authentication channel between end hosts, where keys are not transmitted in the clear. The protocol generates a master secret for a connection, and a method is described in [3] for exporting a key generated from this and other properties which can then be used by the application layer. This document shows how to use this exported key, along with the method in draft-paasch-mptcp-application-authentication, for providing alternative keying mechanisms for MPTCP.

2. Technical Implementation

As described in draft-paasch-mptcp-application-authentication, the initial MP_CAPABLE handshake will exchange an arbitrary token for identifying an MPTCP connection. Whilst it is RECOMMENDED that the token is hard to guess, it can be used to carry any data, such as arbitrary routing information, and the security provided by the application-layer security will mitigate any risks of an attacker guessing tokens.

When an MPTCP end host wishes to open a new subflow, it will follow the same exchange as described in [4], however the keying material (Key-A and Key-B) will be derived from the TLS handshake, as described in [3]. The "label" field MUST be "EXPORTER-MPTCP". The length used in the key-derivation, following [3] is 16. Key-A are the 64 most-significant bits, while Key-B are the 64 remaining bits. This requires the key exchange to have completed before subflows can be created. Other than the source of the keys, the exchange remains the same. The MP_CAPABLE and MP_JOIN exchange therefore looks like this:



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 1: Example Use of MPTCP Authentication

3. Security Considerations

This draft relies on the security provided by TLS [6] and the key export mechanism of [3] to provide additional security for the MPTCP handshake mechanism. These changes remove lingering risks, originally identified in [7], where an intercept of the initial MPTCP handshake could allow session hijack.

4. IANA Considerations

IANA would be requested to add a value to the TLS Exporter Label registry as described in [3]. The label is "EXPORTER-MPTCP".

5. References

5.1. Normative References

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

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

[3] Rescorla, E., "Keying Material Exporters for Transport Layer Security (TLS)", RFC 5705, DOI 10.17487/RFC5705, March 2010, <http://www.rfc-editor.org/info/rfc5705>.

[4] Ford, A., Raiciu, C., Handley, M., Bonaventure, O., and C. Paasch, "TCP Extensions for Multipath Operation with Multiple Addresses", draft-ietf-mptcp-rfc6824bis-05 (work in progress), January 2016.

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

5.2. Informative References

[6] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <http://www.rfc-editor.org/info/rfc5246>.

[7] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, January 2013.

Authors' Addresses

Christoph Paasch
Apple, Inc.
Cupertino
US

EMail: cpaasch@apple.com

Alan Ford
Pexip

EMail: alan.ford@gmail.com

MPTCP Working Group
Internet-Draft
Intended status: Informational
Expires: January 6, 2017

B. Peirens
Proximus
G. Detal
S. Barre
O. Bonaventure
Tessares
July 05, 2016

Link bonding with transparent Multipath TCP
draft-peirens-mptcp-transparent-00

Abstract

This document describes the utilisation of the transparent Multipath TCP mode to enable network operators to provide link bonding services in hybrid access networks.

Status of this Memo

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

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

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

This Internet-Draft will expire on January 6, 2017.

Copyright Notice

Table of Contents

1. Introduction

Internet Service Provider networks are composed of different parts : access networks, metropolitan and wide area networks. Given the growing demand for bandwidth, these networks must evolve. In the metropolitan and wide area parts, bandwidth increases thanks to the utilisation of optical fiber or through link aggregation. Increasing bandwidth in the core is not sufficient to allow all users to benefit from faster services. For many operators, the last-mile of the access network remains a bottleneck that is difficult to upgrade.

Many service providers do not rely on a single access network technology. They often have deployed different access networks that were initially targeted at different types of users and customers. Such access networks include xDSL, DOCSIS, FTTx and various wireless technologies (3G, 4G, Wimax, satellite, 5G, ...). With these different access networks, service providers have different ways to reach their customers and combining two access networks would enable them to deliver higher bandwidth services to their customers [I-D.zhang-banana-problem-statement].

In this document, we first describe in section Section 2 the hybrid access networks that are being deployed by various network operators. We focus on the aggregation of a fixed network (e.g. xDSL) with a cellular network (e.g. LTE). Many operators wish to use the bandwidth that is not consumed by the mobile devices on their cellular network to deliver better services to their fixed line customers. Section Section 3 lists the main requirements expressed by these operators. Section Section 4 briefly evaluates whether the main proposed bonding techniques meet those requirements. We then describe in section Section 5 how a transparent mode of operation for Multipath TCP [RFC6824] can be used to meet those operator requirements.

2. Reference architecture

Our reference architecture is shown in figure Figure 1. We use a similar terminology as in [WT-348] and consider the following elements :

- a single homed end host H that is attached through one interface (e.g. WiFi) to a Hybrid Customer Premisses Equipment (HCPE)
- a Hybrid Customer Premises Equipment (HCPE) that is connected to two different access networks. The solution proposed in this document support any number of access networks, but we restrict our examples to two.
- A Hybrid Aggregation Gateway (HAG) that is reachable over both access networks
- a regular server, S

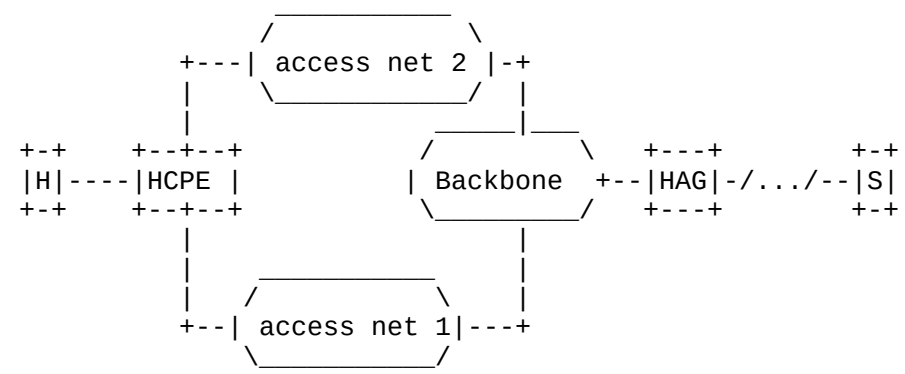


Figure 1: Hybrid access networks

We assume that IP addresses are assigned according to the best current practices, i.e. host H is allocated one IP address, and one IP address is assigned to each interface of the HCPE. Furthermore, BCP 38 [RFC2827] is used on the two access networks attached to the HCPE. The solution proposed in this document is agnostic of the IP version that is used. It operates equally well with both IPv4 and IPv6 and can use any mix of IPv4/IPv6. When writing IP addresses, we use the @ notation. For example, H@ is the IP address assigned to host H, HCPE@1 is the IP address assigned to the HCPE on access network 1,... For most network operators, the different access networks that need to be aggregated are not equivalent. One network, typically a fixed access network managed by the operator, is

considered to be the main access network. The other access network, possibly managed by another network operator, is used to provide additional capacity to cope with bandwidth limitations on the primary access network. We focus on this bandwidth aggregation scenario in this document. While the second access network can also be used in case of failure of the primary access network we currently leave it out of scope of the solution (existing solutions are already deployed by operators for this).

3. Operator requirements

Many operators have expressed their interest in efficiently supporting hybrid access networks. We list here some of the requirements that they have identified and have guided the design of the proposed solution.

- Req1: the bonding solution MUST support IPv6 and IPv4
- Req2: the bonding solution SHOULD minimize the additional delay that it introduces in the network
- Req3: the bonding solution MUST not expose multiple addresses for a given customer and the same address MUST be used for all transport protocols used by each customer
- Req4 : the bonding solution MUST not use more than one public IPv4 address per customer
- Req5 : the bonding solution SHOULD enable the operator to trace the connections created by a given customer
- Req6 : the bonding solution MUST monitor the quality of the different links and adapt the load distribution dynamically according to the load and the operator’s policies
- Req7 : the bonding solution MUST be decoupled from the underlying fixed/mobile access network
- Req8: the bonding solution MUST be able to efficiently load-balance the packets belonging to a single TCP connection over several access networks
- Req9: the bonding solution SHOULD not change the subscriber service attachment and authentication point in the network.

The second requirement reflects the importance of minimising the latency. Many applications, including HTTP, are affected by any increased latency. The third requirement reflects operational issues. Many applications expect that all the flows originated by a host will have the same source address, independently of the protocol used for each flow. A solution that would use different addresses for different transport protocols or for flows that do not benefit from hybrid access (e.g. by defined policy), would cause operational problems. The fifth requirement reflects the desire of the network operator to have some visibility of the flows that pass through its access network in order to apply filtering rules, log flows or provide QoS. The sixth requirement reflects the fact that the

bandwidth of the access networks that are aggregated can vary quickly. This is particularly the case for cellular networks where mobile devices could have priority over the bonding service. The last two requirements correspond to the utilisation of large TCP flows. Measurement studies in access networks show that TCP is the dominant protocol in these networks and that most of the data volume is carried by long TCP connections. Such connections must be load-balanced on a per packet basis to achieve a good aggregation.

4. Existing solutions

In this document, we focus on solutions that can combine very different access network technologies, typically a fixed line access network such as xDSL and a wireless access network such as LTE. We discuss only some of the proposed techniques. A complete overview of all the available solutions is outside the scope of this document.

4.1. Datalink solutions for hybrid access networks

Some datalink technologies, such as Multilink PPP [RFC1990], can load balance packets over different links. Unfortunately, they cannot easily be used in hybrid access networks that rely on different types of datalinks.

4.2. Network layer solutions for hybrid access networks

Various solutions exist in the network layer. A first possibility is to assign the same address to the HCPE (and thus the hosts behind it) over the different access networks. This requires a specific configuration of the routing in the access network and some network operators have deployed such solutions. Per-flow and per-packet load balancing are possible with this approach. Unfortunately, it has a number of important drawbacks :

- it is difficult for the HAG/HCPE to measure the performance of the different access networks in to adjust their utilisation in realtime (Req6)

- assigning the same address to the HCPE over different networks requires integration on the subscriber attachment points for both the fixed and mobile network (e.g. BNG & P-GW) for the bonding solution which might not be desirable (Req7)

- if packets from a transport connection are spread over different access networks, they experience different delays and different levels of congestion, but the transport protocol is unaware of those different networks. The net result is a lower throughput since the congestion control scheme adapts the throughput to the access network offering the lowest performance (Req8).

An alternative to assigning the same IP addresses on the different access networks is to use tunnels between the HCPE and the HAG. Various types of IP tunnels are possible [RFC2784] [I-D.zhang-gre-tunnel-bonding]. With such tunnels, the problems mentioned above remain and the tunneling protocol adds a per-packet overhead which may be significant in some environments. Extensions have been recently proposed to include flow control mechanisms in

some of these tunneling techniques [I-D.zhang-banana-tcp-in-bonding-tunnels] but this increases the complexity of the solution. Tunnel based solutions assign the external exposed customer address within the tunnel and change the subscriber service attachment point (Req9) which forces operators to re-implement authentication, logging and service definitions at a different location than the non-hybrid access customers. See also concerns listed in the next section {#transport}.

4.3. Transport layer solutions

The Multipath TCP plain mode option [I-D.boucadair-mptcp-plain-mode] was recently proposed as a solution to cope with some of the above problems of the network layer solutions. This solution is an extension of the TCP option proposed in [HotMiddlebox13b]. With the plain mode option, the HAG maintains a pool of public addresses that are used to translate the client addresses. From an addressing viewpoint, this is equivalent to the deployment of a carrier-grade NAT which leads to operational problems for the management of access-lists that are used to provide QoS, firewalling, but also for the collection of meta data about customer traffic, logs, ... With [I-D.boucadair-mptcp-plain-mode], all the TCP traffic in the access networks appears to be destined to the HAG.

While the Multipath TCP plain mode optionally allows transparency of the source address by using the option a second time with D-bit set to zero, it would require subscriber session information from the network element that assigned the now embedded source address to correctly implement BCP-38 [RFC2827] validation when restoring this at the HAG.

4.4. Application layer solutions

The SOCKS protocol [RFC1928] was designed to enable clients behind a firewall to establish TCP connections through a TCP proxy running on the firewall. A possible deployment in hybrid access networks is to use the HAG as a SOCKS server over Multipath TCP to benefit from its aggregation capabilities. Since regular hosts usually do not use a SOCKS client and do not support Multipath TCP, the HCPE needs to act as a transparent TCP/Multipath-TCP+SOCK proxy.

Compared with the network layer solutions and [I-D.boucadair-mptcp-plain-mode], the SOCKS approach has several drawbacks from an operational viewpoint :

- the HAG must maintain a pool of public addresses

- to establish a TCP connection through a SOCKS server running on the HAG, the HCPE must first perform the three-way handshake and then exchange SOCKS messages to authenticate the client (the number of messages is function of the SOCKS authentication scheme that is used). This increases the establishment time for each TCP connection by one or more additional round-trip times (Req2).

5. The transparent MPTCP mode

The transparent MPTCP mode proposed in this documented was designed under the assumption that in many hybrid access networks, there is a primary access network and the other access network(s) that are combined are used to (virtually) increase the capacity of the primary access network. In such networks, operators usually expect that the secondary access networks will only be used if the primary access networks does not have sufficient capacity to handle the load.

The solution is targeted at TCP traffic only. Non TCP traffic is sent over the primary access network. Support for other transport protocols over the secondary access networks is outside the scope of this document.

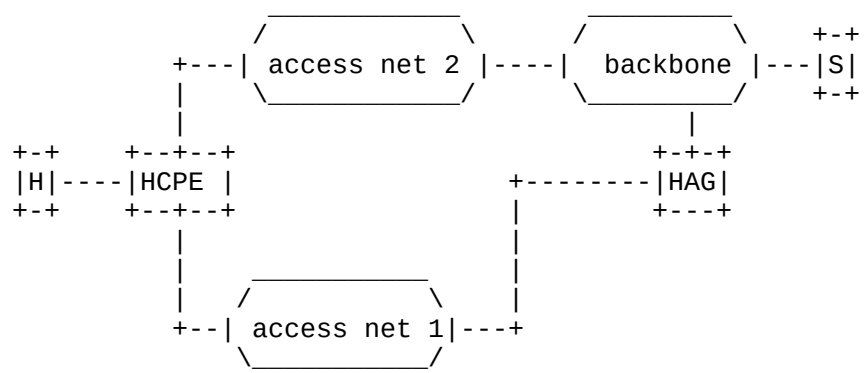


Figure 2: Reference architecture

We consider the network environment shown in figure Figure 2. Access net 1 is the primary network. This figure reflects the specific network configuration that is required for the transparent Multipath TCP mode. The HAG MUST reside on the path followed by the packets sent to/from the HCPE that it serves. This can be achieved, by e.g. using a specific mobile APN that has restricted routing, using service chaining at BNG/BRAS, using specific BNG/BRAS domains or AAA/ RADIUS triggered policy routing at BNG/BRAS. Several vendors have implemented such solutions and they are deployed in various networks.

A HAG typically serves a group of HCPEs and several HAGs can be deployed by an operator. Note that the requirement of placing the HAG on the path of the HCPE that it serves only applies to the primary access network. The other access networks only need to be able to reach the HAG. They do not need direct Internet access.

The HCPE has two IP addresses (or IP prefixes in the case of IPv6

prefix delegation) : HCPE@1 and HCPE@2. HCPE@1 is the primary address prefix assigned to the HCPE and host H uses one address from this prefix as its public address when contacting remote servers (we assume IPv6 in this description. With IPv4, the HCPE will assign a private IPv4 address to the hosts that it serves and will perform NAT). The HAG has one IP address that is reachable from the secondary network, identified as HAG@2. This is illustrated by the vertical link on the HAG in Figure 2.

Both the HCPE and the HAG include a transparent Multipath-TCP/TCP proxy. Various forms of TCP proxies have been defined and are deployed [RFC3135]. The HCPE uses its TCP/Multipath TCP proxy to convert the regular TCP connections initiated by the client host, H, into Multipath TCP connections towards the distant server. However, these Multipath TCP connections do not directly reach the distant server. They are converted into regular TCP connections by the Multipath-TCP/TCP proxy running on the HAG. This is illustrated in figure Figure 3.

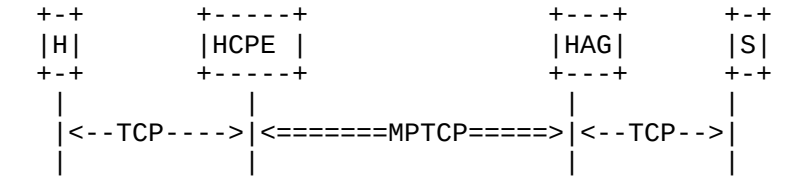


Figure 3: The TCP<->Multipath TCP proxies used on the HCPE and the HAG

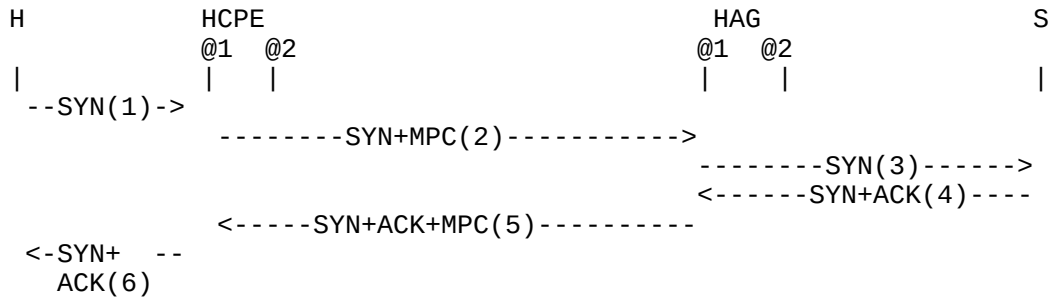


Figure 4: Creation of the initial subflow with the transparent mode

The operation of the transparent mode is illustrated in figure Figure 4. We consider the establishment of one TCP connection from host H (using address H@) to a distant server, S@. The following

packets are exchanged :

- (1) H sends a SYN towards S@.

- (2) The HCPE intercepts the SYN of the initial handshake. It creates some state for a regular TCP connection between H@ and itself acting as a transparent proxy for S@ and a Multipath TCP connection towards S@. These two connections are linked together and any data received over one connection is forwarded over the other. The HCPE then sends a SYN with the MP_CAPABLE option towards S@ over its primary access network to create a Multipath TCP connection to the HAG. Over the primary access network, this SYN appears as originating from H@ and being sent to S@.

- (3) The HAG acts as a transparent proxy for S@ and intercepts the SYN that contains the MP_CAPABLE option. It creates some state for this Multipath TCP connection and initiates a regular TCP connection towards S@. It should be noted that neither the HCPE nor the HAG perform address translation. The distant server receives the SYN from the client as originating from address H@.

- (4) The server replies with a SYN+ACK to confirm the establishment of the connection.

- (5) The HAG intercepts the returning SYN+ACK. The HAG then sends a SYN+ACK with the MP_CAPABLE option to confirm the establishment of the Multipath TCP connection that is proxied by the HCPE.

- (6) The HCPE sends a SYN+ACK to the client host to confirm the establishment of the regular TCP connection

At this point, the establishment of the three connections can be finalised by sending a third ACK. Data can be exchanged by the client and the server through the proxied connections.

The end-to-end connection between the client host (H) and the server (S) is composed of three TCP connections : a regular TCP connection between the host and the HCPE, a Multipath TCP connection between the HCPE and the HAG and a regular TCP connection between the HAG and the remote server. All the packets sent on these three connections contain the H@ and S@ addresses in their IP header.

To use another access network, the HAG simply advertises its address reachable through this access network (HAG@2) on the initial subflow by sending an ADD_ADDR option (1). This triggers the establishment of an additional subflow from the HCPE over the second access network (arrows (2), (3) and (4) in figure Figure 5). The endpoints of this subflow are the IP address of the HCPE on the second access network,

i.e. HCPE@2, and the IP address of the HAG, i.e. HAG@2. Note that the ADD_ADDR option shown in figure Figure 5 is optional. If the HCPE already knows, e.g. by configuration or through mechanisms such as [I-D.boucadair-mptcp-radius] or [I-D.boucadair-mptcp-dhc], the IP address of the HAG, it can crate the additional subflow without waiting for the ADD_ADDR option.

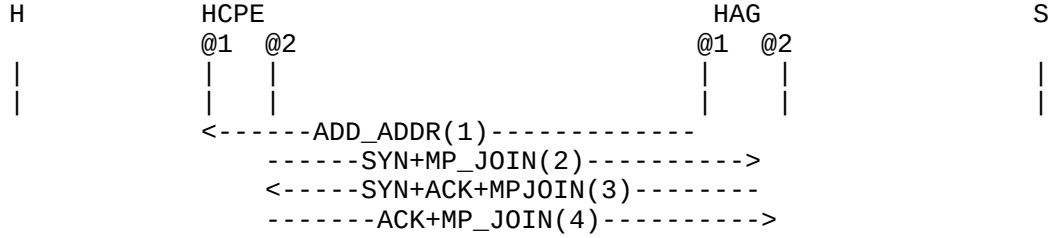


Figure 5: Creation of the second subflow by the HCPE with the transparent MPTCP mode

At this point, any data received from the host by the HCPE or from the server by the HAG can be transported over any of the established subflows. Both the HAG and the HCPE select the most appropriate subflow based on their policies and the current network conditions that are automatically measured by Multipath TCP.

This is not the only way to create additional subflows. The HAG may also create additional subflows. This is illustrated in figure Figure 6 where we assume that the HAG already knows the IP address of the HCPE and thus does not wait for the reception of an ADD_ADDR option from the HCPE to create the additional subflow.

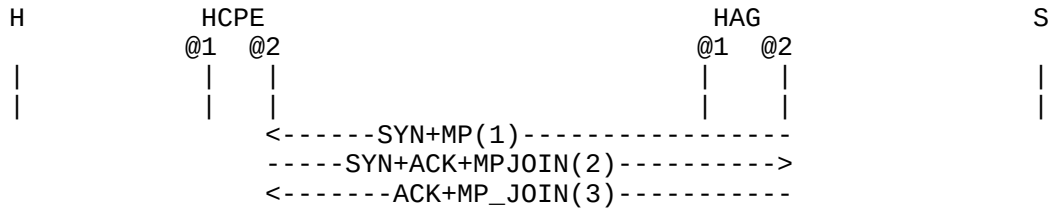


Figure 6: Creation of the second subflow by the HAG with the transparent MPTCP mode

6. Security considerations

Providing a bonding service through different access networks introduces new capabilities, but also new threats to the network. We focus in this section on the threats that are specific to the bonding service and assume that the CPE devices implement the recommendations listed in [RFC6092]. For the HAG, since it operates on the path like a router, many of the the security considerations for routers apply.

When Multipath TCP is used over different paths, the security threats listed in [RFC6181] and [RFC7430] need to be considered. Some of these can be mitigated through proper configuration of the HCPEs, HAGs and access networks.

An important security threat with Multipath TCP is if an attacker were able to inject data on an existing Multipath TCP by associating an additional subflow. Such an attack is already covered by the utilisation of keys in the Multipath TCP handshake. Thanks to the utilisation of the tokens and the HMAC in the MP_JOIN option, the HAG and the HCPE will refuse additional subflows created by an attacker who did not observe the initial SYN packets. Note that since the keys are only exchanged on the first access network, this attacker would have to reside on this access network.

Since the HAG and the HCPE only create subflows among themselves, it is possible for an operator to configure those devices to only accept SYN packets with the MP_CAPABLE or MP_JOIN option to those prefixes. Furthermore, the second access network does not need to be connected to the Internet. This implies that an attacker would need to reside on this network to send packets towards the visible address of the HAG. Ingress filtering and uRPF should be deployed on the access networks to prevent spoofing attacks.

If TCP connections originating from the Internet are accepted on the HCPEs, then the HAG must be secured against denial of service attacks since it will be involved in the processing of all incoming SYN packets.

7. IANA Considerations

There are no IANA considerations in this document.

8. Conclusion

In this document, we have proposed the transparent mode for Multipath TCP and described its utilisation in hybrid access networks where a secondary access network is used to complement a primary access network. Our solution leverages the flow and congestion control capabilities of Multipath TCP to allow an efficient utilisation of the different access networks, even if their capacity fluctuates.

Compared with network layer solutions such as [I-D.zhang-gre-tunnel-bonding], the transparent mode does not introduce any per-packet overhead and does not require any form of network address translation. Compared with the plain mode Multipath TCP proposed in [I-D.boucadair-mptcp-plain-mode], our solution does not require any form of network address translation which has clear operational benefits.

9. References

9.1. Normative References

[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>.

9.2. Informative References

[HotMiddlebox13b] Detal, G., Paasch, C., and O. Bonaventure, "Multipath in the Middle(Box)", HotMiddlebox'13 , December 2013, <http://inl.info.ucl.ac.be/publications/multipath-middlebox>.

[I-D.boucadair-mptcp-dhc] Boucadair, M., Jacquenet, C., and T. Reddy, "DHCP Options for Network-Assisted Multipath TCP (MPTCP)", draft-boucadair-mptcp-dhc-05 (work in progress), May 2016.

[I-D.boucadair-mptcp-plain-mode] Boucadair, M., Jacquenet, C., Behaghel, D., stefano.secci@lip6.fr, s., Henderickx, W., Skog, R., Bonaventure, O., Vinapamula, S., Seo, S., Cloetens, W., Meyer, U., and L. Contreras, "An MPTCP Option for Network-Assisted MPTCP Deployments: Plain Transport Mode", draft-boucadair-mptcp-plain-mode-08 (work in progress), July 2016.

[I-D.boucadair-mptcp-radius] Boucadair, M. and C. Jacquenet, "RADIUS Extensions for Network-Assisted Multipath TCP (MPTCP)", draft-boucadair-mptcp-radius-01 (work in progress), January 2016.

[I-D.zhang-banana-problem-statement] Cullen, M., Leymann, N., Heidemann, C., Boucadair, M., Hui, D., Zhang, M., and B. Sarikaya, "Problem Statement: Bandwidth Aggregation for Internet Access", draft-zhang-banana-problem-statement-02 (work in progress), July 2016.

[I-D.zhang-banana-tcp-in-bonding-tunnels] Zhang, M., Leymann, N., Heidemann, C., and M. Cullen, "Flow Control for Bonding Tunnels", draft-zhang-banana-tcp-in-bonding-tunnels-00 (work in progress), March 2016.

[I-D.zhang-gre-tunnel-bonding]
Leymann, N., Heidemann, C., Zhang, M., Sarikaya, B., and M. Cullen, "Huawei's GRE Tunnel Bonding Protocol", draft-zhang-gre-tunnel-bonding-03 (work in progress), May 2016.

[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, <http://www.rfc-editor.org/info/rfc1928>.

[RFC1990] Sklower, K., Lloyd, B., McGregor, G., Carr, D., and T. Coradetti, "The PPP Multilink Protocol (MP)", RFC 1990, DOI 10.17487/RFC1990, August 1996, <http://www.rfc-editor.org/info/rfc1990>.

[RFC2784] Farinacci, D., Li, T., Hanks, S., Meyer, D., and P. Traina, "Generic Routing Encapsulation (GRE)", RFC 2784, DOI 10.17487/RFC2784, March 2000, <http://www.rfc-editor.org/info/rfc2784>.

[RFC2827] Ferguson, P. and D. Senie, "Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing", BCP 38, RFC 2827, DOI 10.17487/RFC2827, May 2000, <http://www.rfc-editor.org/info/rfc2827>.

[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, <http://www.rfc-editor.org/info/rfc3135>.

[RFC6092] Woodyatt, J., Ed., "Recommended Simple Security Capabilities in Customer Premises Equipment (CPE) for Providing Residential IPv6 Internet Service", RFC 6092, DOI 10.17487/RFC6092, January 2011, <http://www.rfc-editor.org/info/rfc6092>.

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

[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, <http://www.rfc-editor.org/info/rfc7430>.

[WT-348] Broadband Forum, ., "Hybrid Access for Broadband Network", 2014, <http://datatracker.ietf.org/liaison/1355/>.

Authors' Addresses

Bart Peirens
Proximus

Email: bart.peirens@proximus.com

Gregory Detal
Tessares

Email: Gregory.Detal@tessares.net

Sebastien Barre
Tessares

Email: Sebastien.Barre@tessares.net

Olivier Bonaventure
Tessares

Email: Olivier.Bonaventure@tessares.net