```
MPTCP Working Group                                      O. Bonaventure
Internet-Draft                                                UCLouvain
Updates: RFC6824 (if approved)                            July 06, 2015
Intended status: Informational
Expires: January 7, 2016
```

Multipath TCP Address Advertisement
draft-bonaventure-mptcp-addr-00

Abstract

Multipath TCP [RFC6824] defines the ADD_ADDR option to allow a host to announce its addresses to the remote host. In this document we analyze some of the issues with the address advertisement technique defined [RFC6824] and propose some modifications to mitigate these problems. We also show that the reverse DNS could be an excellent alternative to advertise the stable addresses of a server.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Multipath TCP is an extension to TCP [RFC0793] that was specified in [RFC6824]. Multipath TCP was designed with multi-addressed hosts in mind [RFC6182]. A Multipath TCP connection is composed of subflows that are established between any of the addresses of the communicating hosts. [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.

Experience with Multipath TCP shows that these two options allow to support multi-homed or dual-stack servers [TMA2015] and mobile devices [Cellnet12]. While the ADD_ADDR option has been supported in the Linux implementation of Multipath TCP, other implementors have chosen to not support it [I-D.eardley-mptcp-implementations-survey] while still supporting the REMOVE_ADDR option.

In this document, we first analyse in Section 2 several issues with the current ADD_ADDR option as defined in [RFC6824] and [I-D.ietf-mptcp-rfc6824bis]. Then in Section 3 we show how Multipath TCP could leverage the existing DNS to obtain information about the different addresses attached to a server.

2. Issues with ADD_ADDR

A first issue are the security risks if an attacker is able to send spoofed TCP segments that include the ADD_ADDR option. Multipath TCP [RFC6824] defines the ADD_ADDR option shown in Figure 1.

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

Figure 1: The ADD_ADDR option

From a security viewpoint, this option introduces a potential security risk if an attacker is able to send a spoofed ADD_ADDR option. [I-D.ietf-mptcp-rfc6824bis] proposes a new format for this option by placing a truncated HMAC inside the option to authenticate it. The format for this new option (ADD_ADDR2) is shown in Figure 2.

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

Figure 2: The ADD_ADDR2 option

2.1. Usage of the Address Identifiers

A second issue with the ADD_ADDR option is the management of the address identifiers. At first glance, a Multipath TCP implementation could maintain a table of the IP addresses bound to the local host and associate one identifier with each address. When a new IP address is configured, it is added to the table and the index in the table can be used as its identifier. If a local address stops to be bound to the host, the Multipath TCP can extract its identifier from the table and send the REMOVE_ADDR option over all existing Multipath TCP connections. Unfortunately, such a naive implementation is not possible with the current Multipath TCP implementation.

As defined in [RFC6824], the identifiers 0 are assigned to the addresses that were used for the establishment of the initial subflow. This is because the MP_CAPABLE option does not contain any field to encode an address identifier in contrast with the MP_JOIN option.

An annoying consequence of this design choice is that a Multipath TCP implementation must at least remember the identifier of the address that was used to create the initial subflow. It cannot simply rely on the global address table described above because when an address fails, it must be able to send a REMOVE_ADDR with for address identifier 0 if this address was used to create the initial subflow. This forces a Multipath TCP implementation to at least store the address identifier of the initial subflow for each connection.

One suggestion to ease the maintenance of the addresses on a Multipath TCP implementation would be to stop assuming that the address identifier 0 corresponds to the address used to establish the initial subflow. Instead, the implementation should maintain a table of all the addresses that it uses with Multipath TCP and assign one strictly positive identifier to each address. In this case, each address assigned to the host has the same address identifier for all the Multipath TCP connections. When a new address is learned, it is automatically assigned the next available address identifier and can be announced over all existing Multipath TCP connections depending on the policy applied for the address announcements. When an address is not bound anymore to this host, then the same REMOVE_ADDR option can be sent over all Multipath TCP connections.

There is one missing element in the solution discussed above : how to announce the real address identifier that corresponds to the initial subflow. A simple solution to this problem is to use the ADD_ADDR option without an address as shown in Figure 3.

```
                         1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +---------------+---------------+-------+-------+---------------+
    |     Kind      |     Length    |Subtype| IPVer |  Address ID   |
    +---------------+---------------+-------+-------+---------------+
```

Figure 3: The ADD_ADDR option indicating the address identifier of the current subflow

This empty ADD_ADDR option indicates the address identifier of the address on the subflow over which it has been transmitted. It MUST only be used on the initial subflow since on the other subflows the same information is carried reliably in the MP_JOIN option of the SYN segments. The IPVer field of this ADD_ADDR option MUST match the IP version used for the initial subflow.

2.2. Reliability of the ADD_ADDR Option

A third issue with the ADD_ADDR option is that since it is transmitted as a TCP option, it is not delivered reliably [Cellnet12]. If it announces an IPv4 address, the ADD_ADDR option could be inserted inside a segment that carries data and would thus be delivered reliably like the 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_ADDR2 option cannot be placed in the same segment as a DSS option. In these two cases, the ADD_ADDR/ADD_ADDR2 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, some implementations could decide to transmit several copies of the ADD_ADDR option over the same connection.

3. Learning the Addresses bound to a host through the DNS

[RFC6824] defines the ADD_ADDR option as the basic technique to learn the addresses bound to the remote host. Given the importance of learning those addresses, one would expect this technique to be supported by all Multipath TCP implementations. This is not the case, since only the Linux implementation of Multipath TCP supports the ADD_ADDR option [I-D.eardley-mptcp-implementations-survey] as defined in [RFC6824]. The other implementations do not support this option [I-D.eardley-mptcp-implementations-survey]. This design choice was probably motivated by security concerns with this option and also because these implementations assume that only the client creates the subflows and the server is single-homed. In this case, the client (e.g. a smartphone), can create the subflows from any of its own addresses towards the single address of the server.

However, with the deployment of IPv6, the number of dual-stack clients and servers will grow and it will be important for a host that creates a connection towards the IPv4 address of a server to also learn the IPv6 address associated to this particular server. We show in this section that the DNS could be used to distribute the addressing information that is required by Multipath TCP.

There are three possibilities to use to DNS to distribute the list of addresses associated to a given server. A first approach is to use the existing forward DNS and consider that all the ’A’ and ’AAAA’ records associated with a name correspond to the same server and can be used to establish Multipath TCP subflows. Unfortunately, when several records are associated to a DNS name, this is often for load balancing reasons and those records point to the addresses of different hosts. A second approach would be to define a new DNS record that contains the list of the IP addresses associated to a given host. However, this would require to deploy a new type of DNS record. Proposals that were made in the past to define new RR types were not endorsed by the IETF (e.g., one single RR for dual stack hosts [I-D.li-dnsext-ipv4-ipv6] or a distinct RR for IPv4-Embedded IPv6 Address [I-D.boucadair-behave-dns-a64]).

The third approach that we propose in this document is to use the reverse DNS to encode the information about the alternate addresses that are associated to a given host. The reverse DNS tree typically only contains PTR records that associate names to reverse IPv4 or IPv6 addresses. However, nothing prevents the use of the reverse DNS to store A and AAAA records. This is the approach that we recommend. It does not require any change to the DNS protocol and can leverage dynamic updates to the DNS [RFC3007] and DNSSec to authenticate the advertisement of addresses [RFC4034].

As an example, consider the server whose name is mptcp.example.org and which is reachable via the following IP addresses taken from the documentation prefixes [RFC3849] [RFC5737] :

- 192.0.2.10

- 198.51.100.23

- 2001:db8::1234

The forward DNS will contain the following records for this server

```
     mptcp.example.org.    7200    IN  A     192.0.2.10
     mptcp.example.org.    7200    IN  A     198.51.100.23
     mptcp.example.org.    7200    IN      AAAA  2001:db8::1234
```

In addition, the following entries would be added in the reverse DNS.

```
      10.2.0.192.in-addr.arpa. 7200 IN AAAA  2001:db8::1234
      10.2.0.192.in-addr.arpa. 7200 IN A     198.51.100.23


      23.100.51.198.in-addr.arpa. 7200 IN  AAAA  2001:db8::1234
      23.100.51.198.in-addr.arpa. 7200 IN  A     192.0.2.10


      4.3.2.1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
           8.b.d.0.1.0.0.2.ip6.arpa. 7200 IN A  192.0.2.10
      4.3.2.1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
           8.b.d.0.1.0.0.2.ip6.arpa. 7200 IN A  198.51.100.23
```

These reverse records can, of course, be signed with DNSSec [RFC4034].

4. Conclusion

In this document, we have discussed several issues with the advertisement of addresses with the ADD_ADDR and ADD_ADDR2 options in Multipath TCP. Then, we have shown that the reverse DNS can be used by servers to advertise their alternate IP addresses. This does not require any modification to the DNS protocol and could be used by applications that do not want or cannot rely on the ADD_ADDR option.

5. Acknowledgements

This work was partially supported by the FP7-Trilogy2 project. This document was improved thanks to the comments and suggestions received from Fabien Duchene, Benjamin Hesmans and Mohammed Boucadair.

6. Informative References

[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.boucadair-behave-dns-a64]
Boucadair, M. and E. Burgey, "A64: DNS Resource Record for IPv4-Embedded IPv6 Address", draft-boucadair-behave-dns-a64-02 (work in progress), September 2010.

[I-D.eardley-mptcp-implementations-survey]
Eardley, P., "Survey of MPTCP Implementations", draft-eardley-mptcp-implementations-survey-02 (work in progress), July 2013.

[I-D.ietf-mptcp-rfc6824bis]
Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", draft-ietf-mptcp-rfc6824bis-04 (work in progress), March 2015.

[I-D.li-dnsext-ipv4-ipv6]
Li, L., Li, Z., and X. Duan, "DNS Extensions to Support IPv4 and IPv6", draft-li-dnsext-ipv4-ipv6-02 (work in progress), October 2009.

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

[RFC1323] Jacobson, V., Braden, B., and D. Borman, "TCP Extensions for High Performance", RFC 1323, May 1992.

[RFC3007] Wellington, B., "Secure Domain Name System (DNS) Dynamic Update", RFC 3007, November 2000.

[RFC3849] Huston, G., Lord, A., and P. Smith, "IPv6 Address Prefix Reserved for Documentation", RFC 3849, July 2004.

[RFC4034] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "Resource Records for the DNS Security Extensions", RFC 4034, March 2005.

[RFC5737] Arkko, J., Cotton, M., and L. Vegoda, "IPv4 Address Blocks Reserved for Documentation", RFC 5737, January 2010.

[RFC6182] Ford, A., Raiciu, C., Handley, M., Barre, S., and J. Iyengar, "Architectural Guidelines for Multipath TCP Development", RFC 6182, March 2011.

[RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, January 2013.

[TMA2015] Hesmans, B., Hoang Tran-Viet, ., Sadre, R., and O. Bonaventure, "A first look at real Multipath TCP traffic", TMA 2015 , April 2015, <http://inl.info.ucl.ac.be/publications/first-look-real-multipath-tcp-traffic>.

Author’s Address

Olivier Bonaventure
UCLouvain

Email: Olivier.Bonaventure@uclouvain.be

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 20, 2016

M. Boucadair
C. Jacquenet
France Telecom
T. Reddy
Cisco

November 17, 2015

DHCP Options for Network-Assisted Multipath TCP (MPTCP)
draft-boucadair-mptcp-dhc-04

Abstract

One of the promising deployment scenarios for Multipath TCP (MPTCP) is to enable a Customer Premises Equipment (CPE) that is connected to multiple networks (e.g., DSL, LTE, WLAN) to optimize the usage of its network attachments. Because of the lack of MPTCP support at the server side, some service providers consider a network-assisted model that relies upon the activation of a dedicated function called: MPTCP Concentrator.

This document focuses on the explicit deployment scheme where the identity of the MPTCP Concentrator(s) is explicitly configured on connected hosts. This document specifies DHCP (IPv4 and IPv6) options to configure hosts with Multipath TCP (MPTCP) parameters.

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

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, see for example [RFC4908]. This deployment scenario relies on MPTCP proxies located on both the CPE and network sides (Figure 1). The latter plays the role of traffic concentrator. A concentrator terminates the MPTCP sessions established from a CPE, before redirecting traffic into a legacy TCP session.

```
                         IP Network #1
    +------------+        _--------_    +------------+
    |            |       (e.g., LTE )   |            |
    |   CPE      +======================+            |
    | (MPTCP     |       (_        _)   |Concentrator|
    |  Proxy)    |         (_______)    | (MPTCP     |
    |            |                      |  Proxy)    |------> Internet
    |            |                      |            |
    |            |        IP Network #2 |            |
    |            |        _--------_    |            |
    |            |       ( e.g., DSL )  |            |
    |            +======================+            |
    |            |       (_        _)   |            |
    +-----+------+        (_______)     +------------+
          |
   ----CPE network----
          |
       end-nodes
```

Figure 1: "Network-Assisted" MPTCP Design

Both implicit and explicit modes are considered to steer traffic towards an MPTCP Concentrator. This document focuses on the explicit mode that consists in configuring explicitly the reachability information of the MPTCP concentrator on a host.

This document defines DHCPv4 [RFC2131] and DHCPv6 [RFC3315] options that can be used to configure hosts with MPTCP Concentrator IP addresses.

This specification assumes an MPTCP Concentrator is reachable through one or multiple IP addresses. As such, a list of IP addresses can be returned in the DHCP MPTCP option. Also, it assumes the various network attachments provided to an MPTCP-enabled CPE are managed by the same administrative entity.

2. Terminology

This document makes use of the following terms:

- MPTCP Concentrator (or concentrator): refers to a functional element that is responsible for aggregating the traffic of a group of CPEs. This element is located upstream in the network. One or multiple concentrators can be deployed in the network side to assist MPTCP-enabled CPEs to establish MPTCP connections via available network attachments.

On the uplink path, the concentrator terminates the MPTCP connections [RFC6824] received from its customer-facing interfaces and transforms these connections into legacy TCP connections [RFC0793] towards upstream servers.

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

- DHCP refers to both DHCPv4 [RFC2131] and DHCPv6 [RFC3315].
- DHCP client denotes a node that initiates requests to obtain configuration parameters from one or more DHCP servers.
- DHCP server refers to a node that responds to requests from DHCP clients.

3. DHCPv6 MPTCP Option

3.1. Format

The DHCPv6 MPTCP option can be used to configure a list of IPv6 addresses of an MPTCP Concentrator.

The format of this option is shown in Figure 2. As a reminder, this format follows the guidelines for creating new DHCPv6 options (Section 5.1 of [RFC7227]).

```
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     OPTION_V6_MPTCP           |         Option-length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|                          ipv6-address                         |
|                                                               |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|                          ipv6-address                         |
|                                                               |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              ...                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2: DHCPv6 MPTCP option

The fields of the option shown in Figure 2 are as follows:

- Option-code: OPTION_V6_MPTCP (TBA, see Section 8.1)

- Option-length: Length of the ’MPTCP Concentrator IP Address(es)’ field in octets. MUST be a multiple of 16.
- MPTCP Concentrator IPv6 Addresses: Includes one or more IPv6 addresses [RFC4291] of the MPTCP Concentrator to be used by the MPTCP client.

  Note, IPv4-mapped IPv6 addresses (Section 2.5.5.2 of [RFC4291]) are allowed to be included in this option.

To return more than one MPTCP concentrators to the requesting DHCPv6 client, the DHCPv6 server returns multiple instances of OPTION_V6_MPTCP.

3.2. DHCPv6 Client Behavior

Clients MAY request option OPTION_V6_MPTCP, as defined in [RFC3315], Sections 17.1.1, 18.1.1, 18.1.3, 18.1.4, 18.1.5, and 22.7. As a convenience to the reader, we mention here that the client includes requested option codes in the Option Request Option.

The DHCPv6 client MUST be prepared to receive multiple instances of OPTION_V6_MPTCP; each instance is to be treated separately as it corresponds to a given MPTCP Concentrator: there are as many concentrators as instances of the OPTION_V6_MPTCP option.

If an IPv4-mapped IPv6 address is received in OPTION_V6_MPTCP, it indicates that the MPTCP Concentrator has the corresponding IPv4 address.

The DHCPv6 client MUST silently discard multicast and host loopback addresses [RFC6890] conveyed in OPTION_V6_MPTCP.

4. DHCPv4 MPTCP Option

4.1. Format

The DHCPv4 MPTCP option can be used to configure a list of IPv4 addresses of an MPTCP Concentrator. The format of this option is illustrated in Figure 3.

```
       0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |  Code         |     Length    |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      | List-Length   |   List of     |
      +-+-+-+-+-+-+-+-+    MPTCP      |
      / Concentrator IPv4 Addresses   /
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+   ---
      | List-Length   |   List of     |    |
      +-+-+-+-+-+-+-+-+    MPTCP      |   |
      / Concentrator IPv4 Addresses   /    |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+    |
      .             ...               . optional
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+    |
      | List-Length   |   List of     |    |
      +-+-+-+-+-+-+-+-+    MPTCP      |    |
      / Concentrator IPv4 Addresses   /    |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+   ---
```

Figure 3: DHCPv4 MPTCP option

The fields of the option shown in Figure 3 are as follows:

- Code: OPTION_V4_MPTCP (TBA, see Section 8.2);
- Length: Length of all included data in octets. The minimum length is 5.
- List-Length: Length of the "List of MPTCP Concentrator IPv4 Addresses" field in octets; MUST be a multiple of 4.
- List of MPTCP Concentrator IPv4 Addresses: Contains one or more IPv4 addresses of the MPTCP Concentrator to be used by the MPTCP client. The format of this field is shown in Figure 4.
- OPTION_V4_MPTCP can include multiple lists of MPTCP Concentrator IPv4 addresses; each list is treated separately as it corresponds to a given MPTCP Concentrator.

  When several lists of MPTCP Concentrator IPv4 addresses are to be included, "List-Length" and "MPTCP Concentrator IPv4 Addresses" fields are repeated.

```
      0     8     16    24    32    40    48
      +-----+-----+-----+-----+-----+-----+--
      |  a1 |  a2 |  a3 |  a4 |  a1 |  a2 | ...
      +-----+-----+-----+-----+-----+-----+--
           IPv4 Address 1          IPv4 Address 2 ...
```

This format assumes that an IPv4 address is encoded as a1.a2.a3.a4.

Figure 4: Format of the List of MPTCP Concentrator IPv4 Addresses

OPTION_V4_MPTCP is a concatenation-requiring option. As such, the mechanism specified in [RFC3396] MUST be used if OPTION_V4_MPTCP exceeds the maximum DHCPv4 option size of 255 octets.

4.2. DHCPv4 Client Behavior

To discover one or more MPTCP Concentrators, the DHCPv4 client MUST include OPTION_V4_MPTCP in a Parameter Request List Option [RFC2132].

The DHCPv4 client MUST be prepared to receive multiple lists of MPTCP Concentrator IPv4 addresses in the same OPTION_V4_MPTCP; each list is to be treated as a separate MPTCP Concentrator instance.

The DHCPv4 client MUST silently discard multicast and host loopback addresses [RFC6890] conveyed in OPTION_V4_MPTCP.

5. DHCP Server Configuration Guidelines

DHCP servers that support the DHCP MPTCP Concentrator option can be configured with a list of IP addresses of the MPTCP Concentrator(s). If multiple IP addresses are configured, the DHCP server MUST be explicitly configured whether all or some of these addresses refer to:

1. the same MPTCP Concentrator: the DHCP server returns multiple addresses in the same instance of the DHCP MPTCP Concentrator option.

2. distinct MPTCP Concentrators : the DHCP server returns multiple lists of MPTCP Concentrator IP addresses to the requesting DHCP client (encoded as multiple OPTION_V6_MPTCP or in the same OPTION_V4_MPTCP); each list refers to a distinct MPTCP Concentrator.

Precisely how DHCP servers are configured to separate lists of IP addresses according to which MPTCP Concentrator they refer to is out of scope for this document. However, DHCP servers MUST NOT combine the IP addresses of multiple MPTCP Concentrators and return them to the DHCP client as if they were belonging to a single MPTCP Concentrator, and DHCP servers MUST NOT separate the addresses of a single MPTCP Concentrator and return them as if they were belonging to distinct MPTCP Concentrators. For example, if an administrator configures the DHCP server by providing a Fully Qualified Domain Name (FQDN) for a MPTCP Concentrator, even if that FQDN resolves to multiple addresses, the DHCP server MUST deliver them within a single server address block.

DHCPv6 servers that implement this option and that can populate the option by resolving FQDNs will need a mechanism for indicating whether to query A records or only AAAA records. When a query returns A records, the IP addresses in those records are returned in the DHCPv6 response as IPv4-mapped IPv6 addresses.

Since this option requires support for IPv4-mapped IPv6 addresses, a DHCPv6 server implementation will not be complete if it does not query A records and represent any that are returned as IPv4-mapped IPv6 addresses in DHCPv6 responses. The mechanism whereby DHCPv6 implementations provide this functionality is beyond the scope of this document.

For guidelines on providing context-specific configuration information (e.g., returning a regional-based configuration), and information on how a DHCP server might be configured with FQDNs that get resolved on demand, see [I-D.ietf-dhc-topo-conf].

6. Security Considerations

The security considerations in [RFC2131] and [RFC3315] are to be considered.

MPTCP-related security considerations are discussed in [RFC6824].

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 boundaries of the network. In order to prevent exhausting the resources of the concentrator by creating an aggressive number of simultaneous subflows for each MPTCP connection, the administrator should limit the number of allowed subflows per host for a given connection.

Attacks outside the domain can be prevented if ingress filtering is 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.

Traffic theft is also a risk if an illegitimate concentrator is inserted in the path. Indeed, inserting an illegitimate concentrator in the forwarding path allows to intercept traffic 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 (for non-transparent modes) should be enabled.

7. Privacy Considerations

Generic privacy-related considerations are discussed in [I-D.ietf-dhc-anonymity-profile].

The concentrator may have access to privacy-related information (e.g., International Mobile Subscriber Identity (IMSI), link identifier, subscriber credentials, etc.). The concentrator must not leak such sensitive information outside an administrative domain.

8. IANA Considerations

8.1. DHCPv6 Option

IANA is requested to assign the following new DHCPv6 Option Code in the registry maintained in http://www.iana.org/assignments/dhcpv6-parameters:

Option Name	Value
OPTION_V6_MPTCP	TBA

8.2. DHCPv4 Option

IANA is requested to assign the following new DHCPv4 Option Code in the registry maintained in http://www.iana.org/assignments/bootp-dhcp-parameters/:

Option Name	Value	Data length	Meaning
OPTION_V4_MPTCP	TBA	Variable; the minimum length is 5.	Includes one or multiple lists of MPTCP Concentrator IP addresses; each list is treated as a separate MPTCP Concentrator.

9. Acknowledgements

Many thanks to Olivier Bonaventure for the feedback on this document. Olivier suggested to define the option as a name but that design approach was debated several times within the dhc wg.

Thanks to Dan Seibel, Bernie Volz, Niall O’Reilly, Simon Hobson, and Ted Lemon for the feedback on the dhc wg mailing list.

10. References

10.1. 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>.

[RFC2131] Droms, R., "Dynamic Host Configuration Protocol", RFC 2131, DOI 10.17487/RFC2131, March 1997, <http://www.rfc-editor.org/info/rfc2131>.

[RFC2132] Alexander, S. and R. Droms, "DHCP Options and BOOTP Vendor Extensions", RFC 2132, DOI 10.17487/RFC2132, March 1997, <http://www.rfc-editor.org/info/rfc2132>.

[RFC3315] Droms, R., Ed., Bound, J., Volz, B., Lemon, T., Perkins, C., and M. Carney, "Dynamic Host Configuration Protocol for IPv6 (DHCPv6)", RFC 3315, DOI 10.17487/RFC3315, July 2003, <http://www.rfc-editor.org/info/rfc3315>.

[RFC3396] Lemon, T. and S. Cheshire, "Encoding Long Options in the Dynamic Host Configuration Protocol (DHCPv4)", RFC 3396, DOI 10.17487/RFC3396, November 2002, <http://www.rfc-editor.org/info/rfc3396>.

[RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 4291, DOI 10.17487/RFC4291, February 2006, <http://www.rfc-editor.org/info/rfc4291>.

[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>.

10.2. Informative References

[I-D.ietf-dhc-anonymity-profile]
Huitema, C., Mrugalski, T., and S. Krishnan, "Anonymity profile for DHCP clients", draft-ietf-dhc-anonymity-profile-04 (work in progress), October 2015.

[I-D.ietf-dhc-topo-conf]
Lemon, T. and T. Mrugalski, "Customizing DHCP Configuration on the Basis of Network Topology", draft-ietf-dhc-topo-conf-06 (work in progress), October 2015.

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

[RFC4908] Nagami, K., Uda, S., Ogashiwa, N., Esaki, H., Wakikawa, R., and H. Ohnishi, "Multi-homing for small scale fixed network Using Mobile IP and NEMO", RFC 4908, DOI 10.17487/RFC4908, June 2007, <http://www.rfc-editor.org/info/rfc4908>.

[RFC6333] Durand, A., Droms, R., Woodyatt, J., and Y. Lee, "Dual-Stack Lite Broadband Deployments Following IPv4 Exhaustion", RFC 6333, DOI 10.17487/RFC6333, August 2011, <http://www.rfc-editor.org/info/rfc6333>.

[RFC7227] Hankins, D., Mrugalski, T., Siodelski, M., Jiang, S., and S. Krishnan, "Guidelines for Creating New DHCPv6 Options", BCP 187, RFC 7227, DOI 10.17487/RFC7227, May 2014, <http://www.rfc-editor.org/info/rfc7227>.

Authors’ Addresses

Mohamed Boucadair
France Telecom
Rennes 35000
France

Email: mohamed.boucadair@orange.com

Christian Jacquenet
France Telecom
Rennes
France

Email: christian.jacquenet@orange.com

Tirumaleswar Reddy
Cisco Systems, Inc.
Cessna Business Park, Varthur Hobli
Sarjapur Marathalli Outer Ring Road
Bangalore, Karnataka 560103
India

Email: tireddy@cisco.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 11, 2018

M. Boucadair
C. Jacquenet
Orange
T. Reddy
Cisco

October 8, 2017

DHCP Options for Network-Assisted Multipath TCP (MPTCP)
draft-boucadair-mptcp-dhc-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 Conversion Point (MCP). 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. MCPs located in the network are responsible for establishing multi-path communications on behalf of endpoints, thereby taking advantage of MPTCP capabilities to achieve different goals that include (but are not limited to) optimization of resource usage (e.g., bandwidth aggregation), of resiliency (e.g., primary/backup communication paths), and traffic offload management.

This document focuses on the explicit deployment scheme where the identity of the MPTCP Conversion Point(s) is explicitly configured on connected hosts. This document specifies DHCP (IPv4 and IPv6) options to configure hosts with Multipath TCP (MPTCP) parameters.

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

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 relies on MPTCP Conversion Points (MCPs) located on both the CPE and network sides (Figure 1). The latter plays the role of traffic concentrator. An MCP terminates the MPTCP sessions established from a CPE, before redirecting traffic into a legacy TCP session. Further Network-Assisted MPTCP deployment and operational considerations are discussed in [I-D.nam-mptcp-deployment-considerations].

```
  +------------+        _--------_    +----------------+
  |            |       (    LTE   )   |                |
  |   CPE      +=======+          +===+  Backbone      |
  |  (MCP)     |      (_         _)   |   Network      |
  |            |         (_______)    |+--------------+|
  |            |       IP Network #1  || Concentrator ||------> Internet
  |            |                      ||    (MCP)     ||
  |            |                      |+--------------+|
  |            |       IP Network #2  |                |
  |            |        _--------_    |                |
  |            |       (    DSL    )  |                |
  |            +=======+           +==+                |
  |            |      (_         _)   |                |
  +-----+------+        (_______)     +----------------+
        |
  ---- LAN ----
        |
    end-nodes
```

Figure 1: "Network-Assisted" MPTCP Design

This document focuses on the explicit mode that consists in configuring explicitly the reachability information of the MCP on a host. Concretely, the explicit mode has several advantages, e.g.,:

- It does not impose any specific constraint on the location of the MCP. For example, the MCP can be located in any access network, located upstream in the core network, or located in a data canter facility.

- Tasks required for activating the explicit mode are minimal. In particular, this mode does not require any specific routing and/or forwarding policies for handling outbound packets other than ensuring that an MCP is reachable from a CPE, and vice versa (which is straightforward IP routing policy operation).

- The engineering effort to change the location of an MCP for some reason (e.g., to better accommodate dimensioning constraints, to move the MCP to a data canter, to enable additional MCP instances closer to the customer premises, etc.) is minimal

- An operator can easily enforce strategies for differentiating the treatment of MPTCP connections that are directly initiated by an MPTCP-enabled host connected to an MCP if the explicit mode is enabled. Typically, an operator may decide to offload MPTCP connections originated by an MPTCP-enabled terminal from being forwarded through a specific MCP, or decide to relay them via a specific MCP. Such policies can be instructed to the MCP. Implementing such differentiating behavior if the implicit mode is in use may be complex to achieve.

- Multiple MCPs can be supported to service the same CPE, e.g., an MCP can be enabled for internal services (to optimize the delivery of some operator-specific services) while another MCP may be solicited for external services (e.g., access to the Internet). The explicit mode allows the deployment of such scenario owing to the provisioning of an MCP selection policy table that relies upon the destination IP prefixes to select the MCP to involve for an ongoing MPTCP connection, for instance.

- Because the MCP’s reachability information is explicitly configured on the CPE, means to guarantee successful inbound connections can be enabled in the CPE to dynamically discover the external IP address that has been assigned for communicating with remote servers, instruct the MCP to maintain active bindings so that incoming packets can be successfully redirected towards the appropriate CPE, etc.

- Troubleshooting and root cause analysis may be facilitated in the explicit mode since faulty key nodes that may have caused a service degradation are known. Because of the loose adherence to the traffic forwarding and routing polices, troubleshooting a service degradation that is specific to multi-access serviced customers should first investigate the behavior of the involved MCP.

This document defines DHCPv4 [RFC2131] and DHCPv6 [RFC3315] options that can be used to configure hosts with MCP IP addresses.

This specification assumes an MCP is reachable through one or multiple IP addresses. As such, a list of IP addresses can be returned in the DHCP MPTCP option. Also, it assumes the various network attachments provided to an MPTCP-enabled CPE are managed by the same administrative entity.

2. Terminology

This document makes use of the following terms:

- Multipath Conversion Point (MCP): a function that terminates a transport flow and relays all data received over it over another transport flow. This element is located upstream in the network. One or multiple MCPs can be deployed in the network side to assist MPTCP-enabled devices to establish MPTCP connections via available network attachments.

  On the uplink path, the MCP terminates the MPTCP connections [RFC6824] received from its customer-facing interfaces and transforms these connections into legacy TCP connections [RFC0793] towards upstream servers.

  On the downlink path, the MCP turns the legacy server’s TCP connection into MPTCP connections towards its customer-facing interfaces.
- DHCP refers to both DHCPv4 [RFC2131] and DHCPv6 [RFC3315].
- DHCP client denotes a node that initiates requests to obtain configuration parameters from one or more DHCP servers.
- DHCP server refers to a node that responds to requests from DHCP clients.

3. DHCPv6 MPTCP Option

3.1. Format

The DHCPv6 MPTCP option can be used to configure a list of IPv6 addresses of an MCP.

The format of this option is shown in Figure 2. As a reminder, this format follows the guidelines for creating new DHCPv6 options (Section 5.1 of [RFC7227]).

```
       0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |     OPTION_V6_MPTCP           |         Option-length         |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |                                                               |
      |                         ipv6-address                          |
      |                                                               |
      |                                                               |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |                                                               |
      |                         ipv6-address                          |
      |                                                               |
      |                                                               |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |                              ...                              |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2: DHCPv6 MPTCP option

The fields of the option shown in Figure 2 are as follows:

- Option-code: OPTION_V6_MPTCP (TBA, see Section 7.1)
- Option-length: Length of the 'MCP IP Address(es)' field in octets. MUST be a multiple of 16.
- MCP IPv6 Addresses: Includes one or more IPv6 addresses [RFC4291] of the MCP to be used by the MPTCP client.

  Note, IPv4-mapped IPv6 addresses (Section 2.5.5.2 of [RFC4291]) are allowed to be included in this option.

To return more than one MCPs to the requesting DHCPv6 client, the DHCPv6 server returns multiple instances of OPTION_V6_MPTCP. Some guidelines for DHCP servers are elaborated in Appendix A.

3.2. DHCPv6 Client Behavior

Clients MAY request option OPTION_V6_MPTCP, as defined in [RFC3315], Sections 17.1.1, 18.1.1, 18.1.3, 18.1.4, 18.1.5, and 22.7. As a convenience to the reader, we mention here that the client includes requested option codes in the Option Request Option.

The DHCPv6 client MUST be prepared to receive multiple instances of OPTION_V6_MPTCP; each instance is to be treated separately as it corresponds to a given MCP: there are as many MCPs as instances of the OPTION_V6_MPTCP option.

If an IPv4-mapped IPv6 address is received in OPTION_V6_MPTCP, it indicates that the MCP has the corresponding IPv4 address.

The DHCPv6 client MUST silently discard multicast and host loopback addresses [RFC6890] conveyed in OPTION_V6_MPTCP.

4. DHCPv4 MPTCP Option

4.1. Format

The DHCPv4 MPTCP option can be used to configure a list of IPv4 addresses of an MCP. The format of this option is illustrated in Figure 3.

```
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Code         |     Length    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| List-Length   |   List of     |
+-+-+-+-+-+-+-+-+    MPTCP      |
/      MCP IPv4 Addresses       /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  ---
| List-Length   |   List of     |   |
+-+-+-+-+-+-+-+-+    MPTCP      |   |
/      MCP IPv4 Addresses       /   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+   |
.             ...               . Optional
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+   |
| List-Length   |   List of     |   |
+-+-+-+-+-+-+-+-+    MPTCP      |   |
/      MCP IPv4 Addresses       /   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  ---
```

Figure 3: DHCPv4 MPTCP option

The fields of the option shown in Figure 3 are as follows:

- Code: OPTION_V4_MPTCP (TBA, see Section 7.2);
- Length: Length of all included data in octets. The minimum length is 5.
- List-Length: Length of the "List of MCP IPv4 Addresses" field in octets; MUST be a multiple of 4.
- List of MCP IPv4 Addresses: Contains one or more IPv4 addresses of the MCP to be used by the MPTCP client. The format of this field is shown in Figure 4.
- OPTION_V4_MPTCP can include multiple lists of MCP IPv4 addresses; each list is treated separately as it corresponds to a given MCP.

When several lists of MCP IPv4 addresses are to be included, "List-Length" and "MCP IPv4 Addresses" fields are repeated.

```
      0     8     16    24    32    40    48
      +-----+-----+-----+-----+-----+-----+--
      |  a1 |  a2 |  a3 |  a4 |  a1 |  a2 | ...
      +-----+-----+-----+-----+-----+-----+--
           IPv4 Address 1          IPv4 Address 2 ...
```

This format assumes that an IPv4 address is encoded as a1.a2.a3.a4.

Figure 4: Format of the List of MCP IPv4 Addresses

OPTION_V4_MPTCP is a concatenation-requiring option. As such, the mechanism specified in [RFC3396] MUST be used if OPTION_V4_MPTCP exceeds the maximum DHCPv4 option size of 255 octets.

Some guidelines for DHCP servers are elaborated in Appendix A.

4.2. DHCPv4 Client Behavior

To discover one or more MCPs, the DHCPv4 client MUST include OPTION_V4_MPTCP in a Parameter Request List Option [RFC2132].

The DHCPv4 client MUST be prepared to receive multiple lists of MCP IPv4 addresses in the same OPTION_V4_MPTCP; each list is to be treated as a separate MCP instance.

The DHCPv4 client MUST silently discard multicast and host loopback addresses [RFC6890] conveyed in OPTION_V4_MPTCP.

5. Security Considerations

The security considerations in [RFC2131] and [RFC3315] are to be considered.

MPTCP-related security considerations are discussed in [RFC6824].

Means to protect the MCP against Denial-of-Service (DoS) attacks must be enabled. Such means include the enforcement of ingress filtering policies at the boundaries of the network. In order to prevent exhausting the resources of the MCP by creating an aggressive number of simultaneous subflows for each MPTCP connection, the administrator should limit the number of allowed subflows per host for a given connection.

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

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

6. Privacy Considerations

Generic privacy-related considerations are discussed in [RFC7844].

The MCP may have access to privacy-related information (e.g., International Mobile Subscriber Identity (IMSI), link identifier, subscriber credentials, etc.). The MCP must not leak such sensitive information outside an administrative domain.

7. IANA Considerations

7.1. DHCPv6 Option

IANA is requested to assign the following new DHCPv6 Option Code in the registry maintained in http://www.iana.org/assignments/dhcpv6-parameters:

Option Name	Value
OPTION_V6_MPTCP	TBA

7.2. DHCPv4 Option

IANA is requested to assign the following new DHCPv4 Option Code in the registry maintained in http://www.iana.org/assignments/bootp-dhcp-parameters/:

Option Name	Value	Data length	Meaning
OPTION_V4_MPTCP	TBA	Variable; the minimum length is 5.	Includes one or multiple lists of MCP IP addresses; each list is treated as a separate MCP.

8. Acknowledgements

Many thanks to Olivier Bonaventure for the feedback on this document. Olivier suggested to define the option as a name but that design approach was debated several times within the dhc wg.

Thanks to Dan Seibel, Bernie Volz, Niall O’Reilly, Simon Hobson, and Ted Lemon for the feedback on the dhc wg mailing list.

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, <https://www.rfc-editor.org/info/rfc2119>.

[RFC2131] Droms, R., "Dynamic Host Configuration Protocol", RFC 2131, DOI 10.17487/RFC2131, March 1997, <https://www.rfc-editor.org/info/rfc2131>.

[RFC2132] Alexander, S. and R. Droms, "DHCP Options and BOOTP Vendor Extensions", RFC 2132, DOI 10.17487/RFC2132, March 1997, <https://www.rfc-editor.org/info/rfc2132>.

[RFC3315] Droms, R., Ed., Bound, J., Volz, B., Lemon, T., Perkins, C., and M. Carney, "Dynamic Host Configuration Protocol for IPv6 (DHCPv6)", RFC 3315, DOI 10.17487/RFC3315, July 2003, <https://www.rfc-editor.org/info/rfc3315>.

[RFC3396] Lemon, T. and S. Cheshire, "Encoding Long Options in the Dynamic Host Configuration Protocol (DHCPv4)", RFC 3396, DOI 10.17487/RFC3396, November 2002, <https://www.rfc-editor.org/info/rfc3396>.

[RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 4291, DOI 10.17487/RFC4291, February 2006, <https://www.rfc-editor.org/info/rfc4291>.

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

[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, <https://www.rfc-editor.org/info/rfc6890>.

9.2. Informative References

[I-D.nam-mptcp-deployment-considerations] Boucadair, M., Jacquenet, C., Bonaventure, O., Henderickx, W., and R. Skog, "Network-Assisted MPTCP: Use Cases, Deployment Scenarios and Operational Considerations", draft-nam-mptcp-deployment-considerations-01 (work in progress), December 2016.

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

[RFC7227] Hankins, D., Mrugalski, T., Siodelski, M., Jiang, S., and S. Krishnan, "Guidelines for Creating New DHCPv6 Options", BCP 187, RFC 7227, DOI 10.17487/RFC7227, May 2014, <https://www.rfc-editor.org/info/rfc7227>.

[RFC7844] Huitema, C., Mrugalski, T., and S. Krishnan, "Anonymity Profiles for DHCP Clients", RFC 7844, DOI 10.17487/RFC7844, May 2016, <https://www.rfc-editor.org/info/rfc7844>.

[RFC7969] Lemon, T. and T. Mrugalski, "Customizing DHCP Configuration on the Basis of Network Topology", RFC 7969, DOI 10.17487/RFC7969, October 2016, <https://www.rfc-editor.org/info/rfc7969>.

Appendix A. DHCP Server Configuration Guidelines

DHCP servers that support the DHCP MCP option can be configured with a list of IP addresses of the MCP(s). If multiple IP addresses are configured, the DHCP server MUST be explicitly configured whether all or some of these addresses refer to:

1. the same MCP: the DHCP server returns multiple addresses in the same instance of the DHCP MCP option.

2. distinct MCPs : the DHCP server returns multiple lists of MCP IP addresses to the requesting DHCP client (encoded as multiple OPTION_V6_MPTCP or in the same OPTION_V4_MPTCP); each list refers to a distinct MCP.

Precisely how DHCP servers are configured to separate lists of IP addresses according to which MCP they refer to is out of scope for this document. However, DHCP servers MUST NOT combine the IP addresses of multiple MCPs and return them to the DHCP client as if they were belonging to a single MCP, and DHCP servers MUST NOT separate the addresses of a single MCP and return them as if they were belonging to distinct MCPs. For example, if an administrator configures the DHCP server by providing a Fully Qualified Domain Name (FQDN) for an MCP, even if that FQDN resolves to multiple addresses, the DHCP server MUST deliver them within a single server address block.

DHCPv6 servers that implement this option and that can populate the option by resolving FQDNs will need a mechanism for indicating whether to query A records or only AAAA records. When a query returns A records, the IP addresses in those records are returned in the DHCPv6 response as IPv4-mapped IPv6 addresses.

Since this option requires support for IPv4-mapped IPv6 addresses, a DHCPv6 server implementation will not be complete if it does not query A records and represent any that are returned as IPv4-mapped IPv6 addresses in DHCPv6 responses. The mechanism whereby DHCPv6 implementations provide this functionality is beyond the scope of this document.

For guidelines on providing context-specific configuration information (e.g., returning a regional-based configuration), and information on how a DHCP server might be configured with FQDNs that get resolved on demand, see [RFC7969].

Authors’ Addresses

Mohamed Boucadair
Orange
Rennes 35000
France

Email: mohamed.boucadair@orange.com

Christian Jacquenet
Orange
Rennes
France

Email: christian.jacquenet@orange.com

Tirumaleswar Reddy
Cisco Systems, Inc.
Cessna Business Park, Varthur Hobli
Sarjapur Marathalli Outer Ring Road
Bangalore, Karnataka 560103
India

Email: tireddy@cisco.com

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: June 10, 2016

M. Boucadair
C. Jacquenet
France Telecom
D. Behaghel
OneAccess
S. Secci
Universite Pierre et Marie Curie (UPMC)
W. Henderickx
Alcatel-Lucent
R. Skog
Ericsson

December 8, 2015

An MPTCP Option for Network-Assisted MPTCP Deployments: Plain Transport Mode
draft-boucadair-mptcp-plain-mode-06

Abstract

One of the promising deployment scenarios for Multipath TCP (MPTCP) is to enable a Customer Premises Equipment (CPE) that is connected to multiple networks (e.g., DSL, LTE, WLAN) to optimize the usage of its network attachments. Because of the lack of 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 an encapsulation scheme between the CPE and the MPTCP Concentrator. Also, this document specifies how UDP traffic can be distributed among available paths without requiring any encapsulation scheme.

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

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, see for example [I-D.deng-mptcp-proxy] or [RFC4908]. This deployment scenario relies on MPTCP proxies located on both the CPE and network sides (Figure 1). The latter plays the role of traffic concentrator. A concentrator terminates the MPTCP sessions established from a CPE, before redirecting traffic into a legacy TCP session.

```
                         IP Network #1
    +------------+        _--------_    +------------+
    |            |       (e.g., LTE )   |            |
    |   CPE      +=======+          +===+            |
    | (MPTCP     |       (_        _)   |Concentrator|
    |  Proxy)    |         (_______)    | (MPTCP     |
    |            |                      |  Proxy)    |------> Internet
    |            |                      |            |
    |            |        IP Network #2 |            |
    |            |        _--------_    |            |
    |            |       ( e.g., DSL )  |            |
    |            +=======+           +==+            |
    |            |       (_        _)   |            |
    +-----+------+        (_______)     +------------+
          |
   ----CPE network----
          |
       end-nodes
```

Figure 1: "Network-Assisted" MPTCP Design

Both implicit and explicit models are considered to steer traffic towards an MPTCP Concentrator. This document focuses on the explicit model that consists in configuring explicitly the reachability information of the MPTCP concentrator on a host (e.g., [I-D.boucadair-mptcp-dhc]).

This specification assumes an MPTCP Concentrator is reachable through one or multiple IP addresses. Also, it assumes the various network attachments provided to an MPTCP-enabled CPE are managed by the same administrative entity. Additional assumptions are listed in Section 3.

This document explains how a plain transport mode, where 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], SOCKS [RFC1928], etc.), can be enabled.

Also, this document investigates an alternate track where UDP flows can be distributed among available paths without requiring any encapsulation scheme.

The solution in this document does not require the modification of the binding information base (BIB) structure maintained by both the CPE and the Concentrator. Likewise, this approach does not infer any modification of the Network Address Translator (NAT) functions that may reside in both the CPE and the device that embeds the concentrator.

The solution also works properly when NATs are present in the network between the CPE and the Concentrator, unlike solutions that rely upon GRE tunneling. Likewise, the solution accommodates deployments that involve CGN (Carrier Grade NAT) upstream the Concentrator.

2. Terminology

This document makes use of the following terms:

- Customer-facing interface: is an interface of the MPTCP Concentrator that is visible to a CPE and which is used for communication purposes between a CPE and the MPTCP Concentrator.

- 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 can be embedded in a CPE and/or a Concentrator.

- MPTCP leg: Refers to a network segment on which MPTCP is used to establish TCP connections.

- MPTCP Concentrator (or concentrator): refers to a functional element that is responsible for aggregating the traffic of a group of CPEs. This element is located upstream in the network. One or multiple concentrators can be deployed in the network side to assist MPTCP-enabled CPEs to 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 turns the legacy server’s TCP connection into MPTCP connections towards its customer-facing interfaces.

3. Assumptions

The following assumptions are made:

- The logic for mounting network attachments by a host is deployment- and implementation-specific and is out of scope of this document.
- The Network Provider that manages the various network attachments (including the concentrators) can enforce authentication and authorization policies using appropriate mechanisms that are 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 or system-wide.
- The concentrator may be notified about the results of monitoring (including probing) the various network legs to service a customer, a group of customers, a given region, etc. No assumption is made by this document about how these monitoring (including probing) operations are executed.
- An MPTCP-enabled, multi-interfaced 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 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 (even if they may not be present on the different network accesses) designs, etc. Nevertheless, in order to take advantage of MPTCP, the location of the concentrator should not jeopardize packet forwarding performance for traffic sent from or directed to connected hosts.

4. Introducing the MPTCP Plain Transport Mode

4.1. An Alternative to Encapsulation

The design option for aggregating various network accesses often relies upon the use of an encapsulation scheme (such as GRE) between the CPE and the Concentrator. The use of encapsulation is motivated by the need to steer traffic through the concentrator and also to allow the distribution of UDP flows 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 document specifies another approach that relies upon plain transport mode between the CPE and the Concentrator.

The use of a plain transport mode does not require the upgrade of any intermediate function (security, TCP optimizer, etc.) that may be located on-path. Thus, the introduction of MPTCP concentrators in operational networks to operate plain mode does not add any extra complexity as far as the operation of possible intermediate functions is concerned.

4.2. Plain Mode MPTCP Option

The format of the Plain Mode MPTCP option is shown in Figure 2.

```
        0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
       +---------------+---------------+-------+-------+---------------+
       |     Kind      |    Length     |SubType|D|U|       Flag Bits   |
       +---------------+---------------+-------+-------+---------------+
       |          Address (IPv4 - 4 octets / IPv6 - 16 octets)         |
       +-------------------------------+-------------------------------+
       |   Port (2 octets, optional)   |
       +-------------------------------+
```

Figure 2: Plain Mode MPTCP Option

The description of the fields is as follows:

- Kind and Length: are the same as in [RFC6824].
- Subtype: to be defined by IANA (Section 6).
- D-bit (direction bit): This flag indicates whether the enclosed IP address (and a 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.
- U-bit (UDP bit): The use of this flag is detailed in Section 5.
- The "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.

- Address: Includes a source or destination IP address. The address family is determined by the "Length" field.

- Port: May be used to carry a port number.

4.3. Theory of Operation

Plain mode operation is as follows:

(1) The CPE is provisioned with the reachability information of one or several Concentrators (e.g., [I-D.boucadair-mptcp-dhc]).

(2) Outgoing TCP packets that can be forwarded by a CPE along MPTCP subflows are transformed into TCP packets carried over a MPTCP connection. The decision-making process to decide whether a flow should be MPTCP-tagged or not is local to the Concentrator and the CPE. It depends on the policies provisioned by the network provider. As such, the decision-making process is policy-driven, implementation- and deployment-specific.

(3) MPTCP packets are sent using a plain transport mode (i.e., without any encapsulation header).

The source IP address and source port number are those assigned locally by the CPE. 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 filtering ([RFC2827]) is implemented at the boundaries of the networks 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 IP address is copied into a dedicated MPTCP option called Plain Mode MPTCP option (see Section 4.2). Because of the limited TCP option space, it is RECOMMENDED to implement the solution specified in [I-D.ietf-tcpm-tcp-edo]. As a reminder, [I-D.touch-tcpm-tcp-syn-ext-opt] specifies a proposal for TCP SYN extended option space.

A binding entry must be maintained by the CPE for that outgoing packet. This binding entry is instantiated by the NAT and/or the firewall functions embedded in the CPE.

(4) Upon receipt of the packet on the MPTCP leg, the Concentrator extracts the IP address included in the Plain Mode MPTCP Option that it uses as the destination IP address of the packet generated in the TCP leg towards its ultimate destination.

The source IP address and port are those of the Concentrators. A binding entry is instantiated by the Concentrator to record the state.

The concentrator may be configured to behave as either a 1:1 address translator or a N:1 translator where the same address is shared among multiple CPEs. Network Providers should be aware of the complications that may arise if a given IP address/prefix is shared among multiple hosts (see [RFC6967]). Whether these complications apply or not is deployment-specific.

The Concentrator should preserve the same IP address that was assigned to a given CPE for all its outgoing connections when transforming an MPTCP connection into a TCP connection.

(5) For incoming TCP packets that need to be forwarded to a CPE, the Concentrator records the source IP address in a Plain Mode MPTCP Option.

The source IP address is replaced with one of the IP addresses listed in the aforementioned binding information base maintained by the Concentrator (if such a state entry exists) or with one of the Concentrator’s IP addresses.

The destination IP address is replaced with the CPE’s IP address (if the corresponding state entry is found in the Concentrator’s binding table) or with one of the CPE’s IP addresses (that are known by the concentrator using some means that are out of the scope of the document).

4.4. Flow Example

A typical flow exchange is shown in Figure 3.

This example assumes no NAT is located between the CPE and the concentrator.

Because the remote server is not MPTCP-aware, the Concentrator is responsible for preserving the same IP address (conc_@, in the example) for the same CPE even if distinct IP addresses (cpe_@1 and cpe_@2, in the example) are used by the CPE to establish subflows with the Concentrator.

```
                                    +-------+
                                    |DNS    |
        +--------+                  |System |         +------------+
        |  CPE   |                  +-------+         |Concentrator|
        +--------+                      |             +------------+
             |                          |                   |
      DNS    |                          |                   |
    -------->|          DNS Query       |                   |
     Query   |------------------------->|                   |
             |   DNS Reply              |                   |
             |<-------------------------|                   |
             |                                              |
             |                                              |
      src=s_@|src=cpe_@1                         dst=conc_@1|src=conc_@
    -------->|--------Plain Mode MPTCP Option(d_@)--------->|-------->
      dst=d_@|                                              |dst=d_@
                                      ....

             |                                              |
      src=d_@|dst=cpe_@1                         src=conc_@1|src=d_@
    <--------|<-------Plain Mode MPTCP Option(d_@)----------|<-------
      dst=s_@|                                              |dst=conc_@
                                      ....

      src=s_@|src=cpe_@2                         dst=conc_@1|src=conc_@
    -------->|--------Plain Mode MPTCP Option(d_@)--------->|-------->
      dst=d_@|                                              |dst=d_@
                                      ....

    Legend:
      * "--Plain Mode MPTCP Option()->" indicates the packet is sent
        in a plain mode, i.e., without any encapsulation hander,
        and that "Plain Mode MPTCP Option" is carried in the packet.
```

Figure 3: Flow Example (No NAT between the CPE and the Concentrator)

5. UDP Traffic

From an application standpoint, there may be a value to distribute UDP datagrams among available network attachments for the sake of network resource optimisation, for example.

Unlike existing proposals that rely upon encapsulation schemes such as IP-in-IP or GRE, this document suggests the use of MPTCP features to control how UDP datagrams are distributed among existing network attachments. UDP datagrams are therefore transformed into TCP-formatted packets.

The CPE and the Concentrator establish a set of MPTCP subflows. These subflows are used to transport UDP datagrams that are distributed among existent subflows. TCP session tracking may not be enabled for the set of subflows that are dedicated to transport UDP traffic. The establishment of these subflows is not conditioned by the receipt of UDP packets; instead, these subflows are initiated upon CPE reboot or when network conditions change (e.g., whenever a new Concentrator is discovered or a new IP address is assigned to the Concentrator). Additional MPTCP connections may be established to anticipate UDP traffic to be distributed among several paths. The maximum number of MPTCP connections that can be dedicated to UDP traffic may be configured locally to the CPE and the Concentrator. How this parameter is configured is implementation and deployment-specific.

When the CPE (or the Concentrator) transforms a UDP packet into a TCP one, it must insert the Plain Mode MPTCP Option with the U-bit set. When setting the source IP address, the destination IP address, and the IP address enclosed in the Plain Mode MPTCP Option, the same considerations specified in Section 4.3 must be followed.

In addition, the CPE (or the Concentrator) must replace the UDP header with a TCP header. Upon receipt of the packet with the U-bit set, the Concentrator (or the CPE) transforms the packet into a UDP packet and follows the same considerations specified in Section 4.3. Both the CPE and the Concentrator may be configured to disable some features (e.g., reordering). Enabling these features is deployment and implementation-specific.

Relaying UDP packets is not conditioned by TCP session establishment because the required subflows that are dedicated to transport UDP traffic are already in place (either at the CPE or the Concentrator).

A flow example is shown in Figure 4.

```
        +--------+                                    +------------+
        |  CPE   |                                    |Concentrator|
        +--------+                                    +------------+
             | /------------------------------------------\ |
             ||    Dedicated MPTCP SubFlows for UDP        ||
             | \------------------------------------------/ |
             |                                              |
      src=s_@|src=cpe_@1                         dst=conc_@1|src=conc_@
    ---UDP-->|---------------------TCP--------------------->|---UDP-->
      dst=d_@|        Plain Mode MPTCP Option(U,d_@)        |dst=d_@
                                      ....
      src=s_@|src=cpe_@2                         dst=conc_@2|src=conc_@
    ---UDP-->|---------------------TCP--------------------->|---UDP-->
      dst=d_@|        Plain Mode MPTCP Option(U,d_@)        |dst=d_@
             |                                              |
                                      ....
      src=s_@|src=cpe_@1                         dst=conc_@1|src=conc_@
    ---UDP-->|---------------------TCP--------------------->|---UDP-->
     dst=d1_@|        Plain Mode MPTCP Option(U,d_@)        |dst=d1_@
             |                                              |
      src=s_@|src=cpe_@1                         dst=conc_@2|src=conc_@
    ---UDP-->|---------------------TCP--------------------->|---UDP-->
     dst=d1_@|        Plain Mode MPTCP Option(U,d_@)        |dst=d1_@
             |                                              |
```

Figure 4: Distributing UDP packets over multiple paths

6. IANA Considerations

This document requests an MPTCP subtype code for this option:

- Plain Mode MPTCP Option

7. Security Considerations

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.

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 boundaries of the network. In order to prevent exhausting the resources of the concentrator by creating an aggressive number of simultaneous subflows for each MPTCP connection, the administrator should limit the number of allowed subflows per host for a given connection.

Attacks outside the domain can be prevented if ingress filtering is 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.

Traffic theft is also a risk if an illegitimate concentrator is inserted in the path. Indeed, inserting an illegitimate concentrator in the forwarding path allows to intercept traffic 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 (for non-transparent modes) should be enabled.

8. Acknowledgements

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

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>.

[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

[I-D.boucadair-mptcp-dhc] Boucadair, M., Jacquenet, C., and T. Reddy, "DHCP Options for Network-Assisted Multipath TCP (MPTCP)", draft-boucadair-mptcp-dhc-04 (work in progress), November 2015.

[I-D.deng-mptcp-proxy] Lingli, D., Liu, D., Sun, T., Boucadair, M., and G. Cauchie, "Use-cases and Requirements for MPTCP Proxy in ISP Networks", draft-deng-mptcp-proxy-01 (work in progress), October 2014.

[I-D.ietf-tcpm-tcp-edo]
Touch, J. and W. Eddy, "TCP Extended Data Offset Option", draft-ietf-tcpm-tcp-edo-04 (work in progress), November 2015.

[I-D.touch-tcpm-tcp-syn-ext-opt]
Touch, J. and T. Faber, "TCP SYN Extended Option Space Using an Out-of-Band Segment", draft-touch-tcpm-tcp-syn-ext-opt-03 (work in progress), October 2015.

[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>.

[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>.

[RFC4908] Nagami, K., Uda, S., Ogashiwa, N., Esaki, H., Wakikawa, R., and H. Ohnishi, "Multi-homing for small scale fixed network Using Mobile IP and NEMO", RFC 4908, DOI 10.17487/RFC4908, June 2007, <http://www.rfc-editor.org/info/rfc4908>.

[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>.

Authors’ Addresses

Mohamed Boucadair
France Telecom
Rennes 35000
France

Email: mohamed.boucadair@orange.com

Christian Jacquenet
France Telecom
Rennes
France

Email: christian.jacquenet@orange.com

Denis Behaghel
OneAccess

Email: Denis.Behaghel@oneaccess-net.com

Stefano Secci
Universite Pierre et Marie Curie (UPMC)
Paris
France

Email: stefano.secci@lip6.fr

Wim Henderickx
Alcatel-Lucent
Belgium

Email: wim.henderickx@alcatel-lucent.com

Robert Skog
Ericsson

Email: robert.skog@ericsson.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 10, 2017

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

March 9, 2017

Extensions for Network-Assisted MPTCP Deployment Models
draft-boucadair-mptcp-plain-mode-10

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 Conversion Point (MCP). 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. MCPs located in the network are responsible for establishing multi-path communications on behalf of endpoints, thereby taking advantage of MPTCP capabilities to achieve different goals that include (but are not limited to) optimization of resource usage (e.g., bandwidth aggregation), of resiliency (e.g., primary/backup communication paths), and traffic offload management.

This document specifies extensions for Network-Assisted MPTCP deployment models.

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 will expire on September 10, 2017.

Copyright Notice

Table of Contents

1. Introduction

The overall quality of connectivity services can be enhanced by combining several access network links for various purposes - resource optimization, better resiliency, etc. Some transport protocols, such as Multipath TCP [RFC6824], can help achieve such better quality, but failed to be massively deployed so far.

The support of multipath transport capabilities by communicating hosts remains a privileged target design so that such hosts can directly use the available resources provided by a variety of access networks they can connect to. Nevertheless, network operators do not control end hosts while the support of MPTCP by content servers remains close to zero.

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 capabilities by communicating peers. Network-Assisted MPTCP deployment models rely upon MPTCP Conversion Points (MCPs) that act on behalf of hosts so that they can take advantage of establishing communications over multiple paths. MCPs can be deployed in CPEs (Customer Premises Equipment), as well as in the provider’s network. MCPs are responsible for establishing multi-path communications on behalf of endpoints. Further details about the target use cases are provided in Section 3.

Most of the current operational deployments that take advantage of multi-interfaced devices rely upon the use of an encapsulation scheme (such as [I-D.zhang-gre-tunnel-bonding], [TR-348]). 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 to intercept traffic and redirect it towards the appropriate MCP.

Current operational MPTCP deployments by network operators are focused on the forwarding of TCP traffic. 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 6 SOCKS messages may have to be exchanged between two MCPs before actual payload data to be transferred, thereby yielding several tens of milliseconds of extra delay before the connection is established) .

To avoid the burden of encapsulation and additional signalling between MCPs, this document explains how a plain transport mode is enabled, so that packets are exchanged between a device and its upstream MCP 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, unlike the aforementioned SOCKS context.

The solution described in this document also works properly when NATs are present in the communication path between a device and its upstream MCP. In particular, the solution in this document accommodates deployments that involve CGN (Carrier Grade NAT) upstream the MCP.

Network-Assisted MPTCP deployment and operational considerations are discussed in [I-D.nam-mptcp-deployment-considerations].

The plain transport mode is characterized as follows:

- 0-RTT proxy.
- No encapsulation required (no tunnels, whatsoever).
- No out-of-band signaling for each MPTCP subflow is required.
- Targets both on-path and off-path MCPs.
- Avoids interference with native MPTCP connections.
- Assists MPTCP connections even if endpoints are MPTCP-capable.
- Accommodates various deployment contexts, such as those that require the preservation of the source IP address and others characterized by an address sharing design. In particular:

    * This solution is compatible with IPv4/IPv6.
    * This solution does not impose any constraint on the addressing scheme to be used by the client.
    * This solution does not require nor exclude the use of distinct IP prefix pools for network-assisted MPTCP deployments.
    * This solution supports both transparent and non-transparent operations.

2. Terminology

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

This document makes use of the following terms:

- Client: an endhost that initiates transport flows forwarded along a single path. Such endhost is not assumed to support multipath transport capabilities.

- Server: an endhost that communicates with a client. Such endhost is not assumed to support multipath transport capabilities.

- Multipath Client: a Client that supports multipath transport capabilities.

- Multipath Server: a Server that supports multipath transport capabilities. Both the client and the server can be single-homed or multi-homed. However, for the use cases discussed in this document, the number of interfaces available at the endhosts is not relevant.

- Transport flow: a sequence of packets that belong to a unidirectional transport flow and which share at least one common characteristic (e.g., the same destination address). TCP and SCTP flows are composed of packets that have the same source and destination addresses, the same protocol number and the same source and destination ports.

- Multipath Conversion Point (MCP): a function that terminates a transport flow and relays all data carried in the flow into another transport flow.

  MCP is a function that converts a multipath transport flow and relays it over a single path transport flow and vice versa.

3. Target Use Cases

We consider two important use cases in this document. We briefly introduce them in this section and leave the details to Section 6 and Section 7. The first use case is a Multipath Client that interacts with a remote Server through a MCP (Section 3.1). The second use case is a multi-homed CPE that includes a MCP and interacts with a remote Server through a downstream MCP (Section 3.2).

3.1. Multipath Client

In this use case, the Multipath Client would like to take advantage of MPTCP even if the Server does not support MPTCP. A typical example is a smartphone that could use both WLAN and LTE access networks to reach a server in order to achieve higher bandwidth or better resilience.

```
+--+                                      +-----+      +--+
|C |                                      | MCP |      |S |
+--+                                      +-----+      +--+
 |                                           |           |
 |<==================MPTCP Leg==============>|<---TCP -->|
 |                                           |           |

Legend:
    C: Client
  MCP: Multipath Conversion Point
    S: Server
```

Figure 1: Network-assisted MPTCP (Host-based Model)

In reference to Figure 1, the MCP terminates the MPTCP connection established by the client and binds it to a TCP connection towards the remote server. Two deployments of this use case are possible.

A first deployment is when the MCP is on the path between the Multipath Client and the Server. In this case, the MCP can terminate the MPTCP connection initiated by the Client and binds it to a TCP connection that the MCP establishes with the Server. When the MCP is not located on all default forwarding paths, the MPTCP connection must be initiated by using the path where the MCP is located.

A second deployment is when the MCP is not on the path between the Multipath Client and the Server. In this case, the Client must first initiate a connection towards the MCP and request it to initiate a TCP connection towards the Server. This is what the SOCKS protocol performs by exchanging control messages to create appropriate mappings to handle the connection. Unfortunately, this requires additional round-trip-time that affects the performance of the end-to-end data transfer, in particular for short-lived connections.

This document specifies the MP_CONVERT Information Element that is carried in the SYN segment of the initial subflow. This SYN segment is sent towards the MCP. The MP_CONVERT Information Element contains the destination address (and optionally a port number) of the Server. Thanks to this information, the MCP can immediately establish the TCP connection with the Server without any additional round-trip-time, unlike a SOCKS-based MPTCP design.

3.2. Multipath CPE

In this use case, neither the Client nor the Server support MPTCP. Two MCPs are used as illustrated in Figure 2. The upstream MCP is embedded in the CPE while the downstream MCP is located in the provider’s network. The CPE is attached to multiple access networks (e.g., xDSL and LTE). The upstream MCP transparently terminates the TCP connections initiated by the Client and converts them into MPTCP connections.

```
                  Upstream                      Downstream
        +--+      +-----+                         +-----+      +--+
        |H1|      | MCP |                         | MCP |      |RM|
        +--+      +-----+                         +-----+      +--+
         |           |                              |           |
         |<---TCP--->|<========MPTCP Leg===========>|<---TCP--->|
         |           |                              |           |
```

Figure 2: Network-assisted MPTCP (CPE-based Model)

The same considerations detailed in Section 3.1 apply for the insertion of the downstream MCP in an MPTCP connection.

4. The MP_PREFER_PROXY MPTCP Option

The implicit mode assumes that the MCP is located on a default forwarding path (Section 5.2.2 of [I-D.nam-mptcp-deployment-considerations]). In such mode, the first subflow must always be placed over that primary path so that the MCP can intercept MPTCP flows. Once intercepted, the MCP advertises its reachability information by means of MPTCP signals (MP_JOIN or ADD_ADDR).

In order to distinguish native MPTCP connections from proxied ones, a new MPTCP option, called MP_PREFER_PROXY, is defined. This option is meant to inform an on-path MCP that the connection should be proxied. The absence of the MP_PREFER_PROXY option is an indication that the corresponding MPTCP connection is native: an on-path MCP must not be involved in such connection. If no explicit signal is included in the initial SYN message, the MCP cannot distinguish "native" MPTCP connections from "proxied" ones.

4.1. Option Format

The format of the MP_PREFER_PROXY is shown in Figure 3.

```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-----------------------+
|     Kind      |    Length     |Subtype|       Reserved        |
+---------------+---------------+-------+-----------------------+
```

Figure 3: MP_PREFER_PROXY MPTCP Option

- Kind and Length: are the same as those defined in Section 3 of [RFC6824]. The size of this option is 4 bytes.

- Subtype: must be allocated by IANA (Section 9).

- "Reserved" 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.

4.2. Option Processing

The MP_PREFER_PROXY option MUST only appear in the SYN message used to create the initial subflow of a Multipath TCP connection.

If the MP_PREFER_PROXY appears in either a SYN segment that does not include the MP_CAPABLE option or a segment whose SYN flag is unset, it MUST be ignored. An implementation MAY log this event since it likely indicates an operational issue.

The sender inserts the MP_PREFER_PROXY option for MPTCP connections that it wants to be proxied by an on-path MCP. Such insertion is possible only when there is enough space left in the dedicated TCP option space.

Upon receipt of a SYN message with an MP_CAPABLE, the MCP MUST check whether an MP_PREFER_PROXY option is present:

- If no such option is included, the MCP MUST NOT interfere with that MPTCP connection (that is, it must not track this MPTCP connection). Processing subsequent subflows of this connection will be handled directly by the endpoints.

- If the MP_PREFER_PROXY option is present, the MCP MUST track the establishment of the connection. That means that the MCP must be prepared to insert itself for the establishment of subsequent subflows, in particular.

Section 5.2.2.1 of [I-D.nam-mptcp-deployment-considerations] details the use of the MP_PREFER_PROXY option.

5. Supplying Data to MCPs

This section focuses mainly on th explicit mode (Section 5.2.1 of [I-D.nam-mptcp-deployment-considerations]) which assumes that the IP reachability information of an MCP is explicitly configured on a device, e.g., by means of a specific DHCP option [I-D.boucadair-mptcp-dhc].

5.1. The MP_CONVERT Information Element

In order to avoid extra delays when establishing a proxied MPTCP connection, specific information are provided to an MCP during the 3WHS. Such information is meant to help the MCP instantiate the required states to process the connection upstream. The supply of such information is achieved by means of an object called the MP_CONVERT (MC) Information Element (IE). This information element typically carries the source/destination IP addresses and/or port numbers of the used by the source and destination endpoints. Other information may also be supplied to an MCP; future extensions may be defined.

The format of the MP_CONVERT Information Element is shown in Figure 4.

```
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------------------------------------------------------+
|                         Magic Number                          |
+---------------+---------------+---------------------------+-+-+
|     Type      |     Length    |          Reserved         |D|M|
+---------------+---------------+---------------------------+-+-+
|          Address (IPv4 - 4 octets / IPv6 - 16 octets)         |
+-------------------------------+-------------------------------+
|   Port (2 octets, optional)   |
+-------------------------------+
```

Figure 4: MP_CONVERT Information Element

The description of the fields is as follows:

- Magic Number: This field MUST be set to "0xFAA8 0xFAA8" to indicate this is an MP_CONVERT Information Element. This field is meant to unambiguously distinguish any data supplied by an application from the one injected by an MCP. Other magic numbers are considered by the authors (e.g., 64 bits that include in addition to "0xFAA8 0xFAA8" 32 bits to enclose the RFC number).

- Type: This field indicates the type of the MP_CONVERT Information Element. It MUST be set to 0 to indicate this element includes an IP address and, eventually, a port number. Other type values MAY be defined in the future.

- Length: Indicates, in bytes, the length of MP_CONVERT Information Element. The minimum size of this option is 4 bytes.

- "Reserved" 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.

- D-bit (Direction bit): this flag indicates whether the enclosed IP address (and port number) reflects the source or the destination IP address (and port number). 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.

- M-bit (More bit): When the M-bit is unset, it indicates that another MP_CONVERT IE is included. When the M-bit is set, it indicates this is the last MP_CONVERT IE included in the payload; if any data is placed right after this MP_CONVERT IE, it is application data.

- Address: includes a source or destination IP address. The address family is determined by the "Length" field. Concretely, a MP_CONVERT Information Element that carries an IPv4 address has a Length field of 8 bytes (or 10, if a port number is included). A MP_CONVERT Information Element that carries 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). This field is optional.

If the length of MP_CONVERT Information Element is not a multiple of 4 bytes, padding MUST be added to preserve 32 bits boundaries.

5.2. Processing an MP_CONVERT Information Element

The MP_CONVERT Information Element is a variable length object that MUST NOT be used in TCP segments whose SYN flag is unset. This IE can only appear in the TCP control messages with SYN flag set. The information carried in the MP_CONVERT IE is used by an MCP to create the initial subflow of a Multipath TCP connection (see the example in Figure 5).

Up to two MP_CONVERT Information Elements with type set to zero can appear inside a SYN segment. If two MP_CONVERT Information Elements with type zero are included, these options MUST NOT have the same D-bit value.

```
          +----+                               +-----+       +--+
          |  C |                               | MCP |       |S |
          +----+                               +-----+       +--+
           |  | ________________________________|             |
           |  /       Initial subflow           \             |
           |  |========SYN(MP_CAPABLE+MC(S))===>|             |
           |  |                                 |--SYN------->|
           |  |                                 |<--SYN/ACK---|
           |  |<====SYN/ACK(MP_CAPABLE)=========|             |
           |  |             ...                 |             |
           |  \ ________________________________/             |
                           ....                     ....
           |  | ________________________________|             |
           |  /       Additional subflow        \             |
           |  \ ________________________________/             |

          Legend:
               <===>: MPTCP leg
               <--->: TCP leg
                MC(): MP_CONVERT Information Element
```

Figure 5: Carrying the MP_CONVERT Information Element

The MP_CONVERT Information Element MUST be included in the payload of a TCP segment whose SYN flag is set.

If the MP_CONVERT Information Element appears in either a SYN segment that does not include the MP_CAPABLE option or a segment whose SYN flag is reset, it MUST be ignored. An implementation MAY log this event since it likely indicates an operational issue.

If the original SYN message contains data in its payload (e.g., [RFC7413]), that data MUST be placed right after the MP_CONVERT IEs when generating the SYN in the MPTCP leg.

An implementation MUST ignore MP_CONVERT Information Elements that include multicast, broadcast, and host loopback addresses [RFC6890]. Concretely, an implementation that receives an MP_CONVERT Information Element with such addresses MUST silently tear down the MPTCP connection.

An implementation that supports the MP_CONVERT Information Element with type zero MUST echo in the SYN/ACK the instances of the MP_CONVERT Information Elements included in a SYN received from the sender. A sender that does not receive in a SYN/ACK a copy of the MP_CONVERT Information Elements it included in a SYN message MUST terminate the MPTCP connection and falls back to TCP or native MPTCP connection. Furthermore, the sender MUST add an entry to its local cache to record the MCPs that do not support the MP_CONVERT Information Element. This cache MUST be flushed out under the following conditions: a new network attachment is detected by the host, a new MCP is configured, the host gets a new IP address/prefix, or a TTL has expired. Subsequent connections to an MCP in the cache MUST NOT be placed using the explicit proxy mode. This procedure is denoted as MCP capability discovery.

In the following sections, MP_CONVERT Information Element is used to refer to the MP_CONVERT Information Element with the type field set to zero. Future documents will specify the exact behavior of processing MP_CONVERT Information Elements with a non zero type field.

6. MPTCP Connections from a Multipath TCP Client

6.1. Description

The simplest usage of the MP_CONVERT Information Element is when a Multipath TCP Client wants to use MPTCP to efficiently utilise different network paths (e.g., WLAN and LTE from a smartphone) to reach a server that does not support Multipath TCP. The basic operation is illustrated in Figure 6.

To use its multipath capabilities to establish an MPTCP connection over the available networks, the Client splits its end-to-end connection towards the TCP Server into two:

(1) An MPTCP connection, that typically relies upon the establishment of one subflow per network path, is established between the client and the MCP.

(2) A TCP connection that is established by the MCP with the server.

Any data that is eligible to be transported over the MPTCP connection is sent by the Client towards the MCP over the MPTCP connection. The MCP then forwards these data over the regular TCP connection until they reach the server. The same forwarding principle applies for the data sent by the Server over the TCP connection with the MCP.

```
        C <===========>MCP <------------> S
        +<============>+

   Legend:
     <===>: subflows of the upstream MPTCP connection
     <--->: downstream TCP connection
```

Figure 6: A Multipath TCP Client interacts with a Server through a Multipath Conversion Point

6.2. Theory of Operation

We assume in this section that the Multipath TCP Client has been configured with the IP address of one or more MCPs which convert the Multipath TCP connection into a regular TCP connection. The address of such MCPs can be statically configured on the Client, dynamically provisioned to the MPTCP Client by means of a DHCP option [I-D.boucadair-mptcp-dhc], or by any other means that are outside the scope of this document.

Conceptually, the MCP acts as a relay between an upstream MPTCP connection and a downstream TCP connection. The MCP has at least a single IP address that is reachable from the Multipath TCP Client. It may be assigned other IP addresses. For the sake of simplicity, we assume in this section that the MCP has a single IP address denoted MCP@. Similarly, we assume that the client has two addresses C@1 and C@2 while address S@ is assigned to the server.

The MCP maps an upstream MPTCP connection (and its associated subflows) onto a downstream TCP connection. On the MCP, an established Multipath TCP connection can be identified by the local Token that was assigned upon reception of the SYN segment.

This Token is guaranteed to be unique on the MCP (provided that it has a single IP address) during the entire lifetime of the MPTCP connection. The 4-tuple (IP src, IP dst, Port src, Port dst) is used to identify the downstream TCP connection.

To initiate a connection to a remote server S, the Multipath TCP Client sends a SYN segment towards the MCP that includes the MP_CONVERT Information Element described in Figure 4. The destination address of the SYN segment is the IP address of the MCP. The MP_CONVERT Information Element included in the SYN contains the IP address and optionally the destination port of the Server (see Figure 7).

```
            +----+                              +-----+     +--+
            |  C |                              | MCP |     |S |
            +----+                              +-----+     +--+
            C@1 C@2                              MCP@       S@
             |  | ________________________________|          |
             |  /       Initial subflow           \          |
             |  |=======SYN(MP_CAPABLE+MC(S@))===>|          |
             |  |                                 |--SYN---->|
             |  |                                 |<-SYN/ACK-|
             |  |<====SYN/ACK(MP_CAPABLE)=========|          |
             |  |             ...                 |          |
             |  \ ________________________________/          |
                             ....                    ....
             |  |________________________________ |          |
             | /       Additional subflow        \|          |
             | \ ________________________________/           |

            Legend:
                 <===>: MPTCP leg
                 <--->: TCP leg
```

Figure 7: Single-ended MCP Flow Example

The MCP processes this SYN segment as follows. First, it generates the local key and a unique Token for the Multipath TCP connection. This Token identifies the MPTCP connection. It is passed to the MCP together with the contents of the MP_CONVERT Information Element (i.e., the address of the destination server) and the destination port.

The MCP then establishes a TCP connection with the destination server. If the received MP_CONVERT Information Element contains a port number, it is used as the destination port of the outgoing TCP connection that is being established by the MCP. Otherwise, the destination port of the upstream MPTCP connection is used as the destination port of the downstream TCP connection. The MCP creates a flow entry for the downstream TCP connection and maps the upstream MPTCP connection onto the downstream TCP connection.

The downstream TCP connection is considered to be active upon reception of the SYN/ACK segment sent by the destination server. The reception of this segment triggers the MCP that confirms the establishment of the upstream MPTCP connection by sending a SYN/ACK segment towards the Multipath TCP Client (including MP_Convert).

At this point, there are two established connections. The endpoints of the upstream Multipath TCP connection are the Multipath TCP Client and the MCP. The endpoints of the downstream TCP connection are the MCP and the Server. These two connections are bound by the MCP.

All the techniques defined in [RFC6824] can be used by the upstream Multipath TCP connection. In particular, the subflows established over the different network paths can be controlled by either the Multipath TCP Client or the MCP. It is likely that the network operators that deploy MCPs will define policies for the utilisation of the MCP. These policies are discussed in Section 5.6 of [I-D.nam-mptcp-deployment-considerations].

Any data received by the MCP on the upstream Multipath TCP connection will be forwarded by the MCP over the bound downstream TCP connection. The same applies for data received over the downstream TCP connection which will be forwarded by the MCP over the upstream Multipath TCP connection.

One of the functions of the MCP is to maintain the binding between the upstream Multipath TCP connection and the downstream TCP connection. If the downstream TCP connection fails for some reason (excessive retransmissions, reception of a RST segment, etc.), then the MCP SHOULD force the teardown of the upstream Multipath TCP connection by transmitting a FASTCLOSE. Similarly, if the upstream Multipath TCP connection fails for some reason (e.g., reception of a FASTCLOSE), the MCP SHOULD tear the downstream TCP connection down and remove the flow entries.

The same reasoning applies when the upstream Multipath TCP connection ends with the transmission of DATA_FINs. In this case, the MCP SHOULD also terminate the bound downstream TCP connection by using FIN segments. If the downstream TCP connection terminates with the exchange of FIN segments, the MCP SHOULD initiate a graceful termination of the bound upstream Multipath TCP connection.

An MCP SHOULD associate a lifetime with the Multipath TCP and TCP flow entries. In this case, it SHOULD use the same lifetime for each pair of bounded connections.

7. MPTCP Connections Between Single Path Client and Server

7.1. Description

There are situations where neither the client nor the server can use multipath transport protocols albeit network providers would want to optimize network resource usage by means of multi-path communication techniques. Hybrid access service offerings are typical business incentives for such situations, where network operators combine a fixed network (e.g., xDSL) with a wireless network (e.g., LTE). In this case, as illustrated in Figure 8, two MCPs are used for each flow. The first MCP, located downstream of the client, converts the single path TCP connection originated from the client into a Multipath TCP connection established with a second MCP. The latter will then establish a TCP connection with the destination server.

```
          Upstream        Downstream
     C <---> MCP <===========> MCP <------------> S
               +<=============>+

Legend:
     <===>: MPTCP leg
     <--->: TCP leg
```

Figure 8: A Client interacts with a Server through an upstream and a downstream Multipath Conversion Points

7.2. Theory of Operation

7.2.1. Downstream MCP

The downstream MCP can be deployed on-path or off-path. If the downstream MCP is deployed off-path, its behavior is described in Section 6.2.

If the downstream MCP is deployed on-path, it only terminates MPTCP connections that carry an empty MP_PREFER_PROXY option inside their SYN (i.e., no address is conveyed). If the MCP receives a SYN segment that contains the MP_CAPABLE option but no MP_PREFER_PROXY, it MUST forward the SYN to its final destination without any modification.

7.2.2. Upstream MCP

The upstream and downstream MCPs cooperate. The upstream MCP may be configured with the addresses of downstream MCPs. If the downstream MCP is deployed on-path, the upstream MCP inserts an MP_PREFER_PROXY option.

In this section, we assume that the upstream MCP has been configured with one address of the downstream MCP. This address can be configured statically, dynamically distributed by means of a DHCP option [I-D.boucadair-mptcp-dhc], or by any other means that are outside the scope of this document.

We assume that the upstream MCP has two addresses uMCP@1 and uMCP@2 while the downstream MCP is assigned a single IP address dMCP@.

The upstream MCP maps an upstream TCP connection onto a downstream MPTCP connection (and its associated subflows) . On the upstream MCP, an established MPTCP connection can be identified by the local Token that was assigned upon reception of the SYN segment from the Client.

The Client sends a SYN segment addressed to the Server and it is intercepted by the upstream MCP which in turns initiates an MPTCP connection towards its downstream MCP that includes the MP_CONVERT Information Element described in Figure 4. The destination address of the SYN segment is the IP address of the downstream MCP. The MP_CONVERT Information Element included in the SYN contains the IP address and optionally the destination port of the Server; this information is extracted from the SYN message received over the upstream TCP connection.

Concretely, the upstream MCP processes the SYN segment received from the Client as follows.

First, it generates the local key and a unique Token for the Multipath TCP connection to identify the MPTCP connection. It extracts the destination IP address and, optionally, the destination port that will then be carried in a MP_CONVERT Information Element. The upstream MCP establishes an MPTCP connection with the downstream MCP. The upstream MCP creates a flow entry for the downstream MPTCP connection and maps the upstream TCP connection onto the downstream MPTCP connection.

The downstream MPTCP connection is considered to be active upon reception of the SYN+ACK segment from the downstream MCP. The reception of this segment triggers the upstream MCP that confirms the establishment of the upstream TCP connection by sending a SYN+ACK segment towards the TCP Client.

At this point, there are two established connections maintained by the upstream MCP:

(1) The endpoints of the upstream TCP connection are the Client and the upstream MCP.

(2) The endpoints of the downstream MPTCP connection are the upstream MCP and the downstream MCP.

These two connections are bound by the upstream MCP. An example is shown in Figure 9.

```
             Upstream                     Downstream
   +--+      +-----+                        +-----+      +--+
   |C1|      | MCP |                        | MCP |      |S1|
   +--+      +-----+                        +-----+      +--+
    C@1   uMCP@1 uMCP@2                       dMCP@       S@
     |         | |______________________________|          |
     |--SYN--->|/       Initial subflow         \          |
     |         |=======SYN(MP_CAPABLE+MC(S@))==>|          |
     |         |                                |--SYN---->|
     |         |                                |<-SYN/ACK-|
     |         |<====SYN/ACK(MP_CAPABLE)========|          |
     |<SYN/ACK-|              ...               |          |
     |          \ ______________________________/          |
                            ....                    ....
     |         | | ____________________________ |          |
     |         | |/       Additional subflow   \|          |
     |         | |\ ___________________________/|          |
                                ....
```

Figure 9: Dual-Ended MCP Flow Example

All the techniques defined in [RFC6824] can be used by the MPTCP connection. In particular, the utilisation of the different network paths can be controlled by one MCP or the other.

Any data received by the upstream MCP over the upstream TCP connection will be forwarded by the MCP over the bound downstream MPTCP connection, assuming such data are eligible to MPTCP transport. The same applies for data received over the downstream MPTCP connection which will be forwarded by the upstream MCP over the upstream TCP connection.

The same considerations as in Section 6.2 apply for the maintenance of the connections by the upstream MCP.

8. Interaction with TFO

This section discusses the implications of using MP_CONVERT Information Elements with TCP Fast Open (TFO). We distinguish between TFO negotiation (i.e., a Fast Open option with an empty cookie field to request a cookie) and TFO data (i.e., SYN with data and the cookie in the Fast Open option).

This section focuses on the implications of using MP_CONVERT Information Element on TFO efficiency. Implications related to MPTCP options and TFO negotiation are not specific to this document; the reader may refer to [I-D.barre-mptcp-tfo].

Distinct implications are assessed depending whether TFO negotiation and usage occurs before MCP capability discovery phase is completed or not (Section 5.2). Concretely, the following cases are discussed:

1. MCP capability discovery was already completed prior to receiving a message with TFO negotiation or TFO data: For this case, the host has already contacted its MCP in the context of a prior connection. The outcome of such connections is used to determine the capabilities of its MCP (Section 5.2).

    A. The MCP supports MP_CONVERT Information Element: Any information provided to an MCP to facilitate MPTCP operation is unambiguously distinguished from TFO data that are also included in the SYN payload. An upstream MCP will remove the MP_CONVERT Information Elements before relaying the SYN message (with TFO data) to the next hop.

    B. The MCP does not support MP_CONVERT Information Element: No additional issue is raised for obvious reasons.

2. MCP capability discovery is not completed prior to receiving a message with TFO negotiation or TFO data.

    A. If the same message is used to negotiate TFO and to retrieve the capabilities of the MCP, extra delay may be observed before negotiating TFO if the MCP does not support the MP_CONVERT Information Element. Obviously, no concern is raised when the MCP supports the MP_CONVERT Information Element.

    B. If the same message includes TFO data and is used to retrieve the capabilities of the MCP, extra delay may be observed before negotiating TFO if the MCP does not support the MP_CONVERT Information Element. Obviously, no concern is raised when the MCP supports the MP_CONVERT Information Element.

To mitigate cases where extra delays are experienced when TFO is present, it is RECOMMENDED to not proxy connections with TFO before the MCP capability discovery procedure is completed.

9. IANA Considerations

This document requests an MPTCP subtype code for this option:

- MP_PREFER_PROXY

10. Security Considerations

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

10.1. Privacy

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

10.2. Denial-of-Service (DoS)

Means to protect the MCP 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 MCP resources by establishing a great number of simultaneous subflows for each MPTCP connection, the MCP 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 an MCP instance are yet another actual threat. Means to ensure that illegitimate nodes cannot connect to a network should be implemented.

10.3. Illegitimate MCP

Traffic theft is a risk if an illegitimate MCP is inserted in the path. Indeed, inserting an illegitimate MCP 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 an MCP should be enabled.

11. Acknowledgements

Many thanks to Chi Dung Phung, Mingui Zhang, Rao Shoaib, Yoshifumi Nishida, and Christoph Paasch for their valuable 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 inputs.

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.

The design approach adopted in -10 is the outcome of fruitful discussions with Alan Ford. Many thanks Alan.

12. References

12.1. Normative References

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

[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>.

12.2. Informative References

[I-D.barre-mptcp-tfo]
Barre, S., Detal, G., and O. Bonaventure, "TFO support for Multipath TCP", draft-barre-mptcp-tfo-01 (work in progress), January 2015.

[I-D.boucadair-mptcp-dhc]
Boucadair, M., Jacquenet, C., and T. Reddy, "DHCP Options for Network-Assisted Multipath TCP (MPTCP)", draft-boucadair-mptcp-dhc-06 (work in progress), October 2016.

[I-D.nam-mptcp-deployment-considerations]
Boucadair, M., Jacquenet, C., Bonaventure, O., Henderickx, W., and R. Skog, "Network-Assisted MPTCP: Use Cases, Deployment Scenarios and Operational Considerations", draft-nam-mptcp-deployment-considerations-01 (work in progress), December 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-05 (work in progress), December 2016.

[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>.

[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>.

[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>.

[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>.

[TR-348] BBF, "Hybrid Access Broadband Network Architecture", July 2016.

Authors’ Addresses

Mohamed Boucadair (editor)
Orange
Rennes 35000
France

Email: mohamed.boucadair@orange.com

Christian Jacquenet (editor)
Orange
Rennes
France

Email: christian.jacquenet@orange.com

Olivier Bonaventure (editor)
Tessares
Belgium

Email: olivier.bonaventure@tessares.net

Denis Behaghel
OneAccess

Email: Denis.Behaghel@oneaccess-net.com

Stefano Secci
UPMC

Email: stefano.secci@lip6.fr

Wim Henderickx (editor)
Nokia/Alcatel-Lucent
Belgium

Email: wim.henderickx@alcatel-lucent.com

Robert Skog (editor)
Ericsson

Email: robert.skog@ericsson.com

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

Bart Peirens
Proximus

Email: bart.peirens@proximus.com

MPTCP Working Group
Internet-Draft
Intended status: Informational
Expires: April 21, 2016

O. Bonaventure
UCLouvain
C. Paasch
Apple, Inc.
G. Detal
UCLouvain and Tessares
October 19, 2015

Use Cases and Operational Experience with Multipath TCP
draft-ietf-mptcp-experience-03

Abstract

This document discusses both use cases and operational experience with Multipath TCP in real world networks. It lists several prominent use cases for which Multipath TCP has been considered and is being used. It also gives insight to some heuristics and decisions that have helped to realize these use cases.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Multipath TCP was standardized in [RFC6824] and five independant implementations have been developed [I-D.eardley-mptcp-implementations-survey]. As of September 2015, Multipath TCP has been or is being implemented on the following platforms :

- Linux kernel [MultipathTCP-Linux]

- Apple iOS and MacOS [Apple-MPTCP]

- Citrix load balancers

- FreeBSD [FreeBSD-MPTCP]

- Oracle

The first three implementations [I-D.eardley-mptcp-implementations-survey] are known to interoperate. The last two are currently being tested and improved against the Linux implementation. Three of these implementations are open-source. Apple’s implementation is widely deployed.

Since the publication of [RFC6824], experience has been gathered by various network researchers and users about the operational issues that arise when Multipath TCP is used in today’s Internet.

When the MPTCP working group was created, several use cases for Multipath TCP were identified [RFC6182]. Since then, other use cases have been proposed and some have been tested and even deployed. We describe these use cases in Section 2.

Section 3 focuses on the operational experience with Multipath TCP. Most of this experience comes from the utilisation of the Multipath TCP implementation in the Linux kernel [MultipathTCP-Linux]. This open-source implementation has been downloaded and is used by thousands of users all over the world. Many of these users have provided direct or indirect feedback by writing documents (scientific articles or blog messages) or posting to the mptcp-dev mailing list (see https://listes-2.sipr.ucl.ac.be/sympa/arc/mptcp-dev). This Multipath TCP implementation is actively maintained and continuously improved. It is used on various types of hosts, ranging from smartphones or embedded routers to high-end servers.

The Multipath TCP implementation in the Linux kernel is not, by far, the most widespread deployment of Multipath TCP. Since September 2013, Multipath TCP is also supported on smartphones and tablets running iOS7 [IOS7]. There are likely hundreds of millions of Multipath TCP enabled devices. However, this particular Multipath TCP implementation is currently only used to support a single application. Unfortunately, there is no public information about the lessons learned from this large scale deployment.

Section 3 is organized as follows. Supporting the middleboxes was one of the difficult issues in designing the Multipath TCP protocol. We explain in Section 3.1 which types of middleboxes the Linux Kernel implementation of Multipath TCP supports and how it reacts upon encountering these. Section 3.2 summarises the MPTCP specific congestion controls that have been implemented. Section 3.3 and Section 3.7 discuss heuristics and issues with respect to subflow management as well as the scheduling across the subflows. Section 3.8 explains some problems that occurred with subflows having different Maximum Segment Size (MSS) values. Section 3.9 presents issues with respect to content delivery networks and suggests a solution to this issue. Finally, Section 3.10 documents an issue with captive portals where MPTCP will behave suboptimally.

2. Use cases

Multipath TCP has been tested in several use cases. There is already an abundant scientific literature on Multipath TCP [MPTCPBIB]. Several of the papers published in the scientific literature have identified possible improvements that are worth being discussed here.

2.1. Datacenters

A first, although initially unexpected, documented use case for Multipath TCP has been in datacenters [HotNets][SIGCOMM11]. Today's datacenters are designed to provide several paths between single-homed servers. The multiplicity of these paths comes from the utilization of Equal Cost Multipath (ECMP) and other load balancing techniques inside the datacenter. Most of the deployed load balancing techniques in datacenters rely on hashes computed over the five tuple. Thus all packets from the same TCP connection follow the same path and so are not reordered. The results in [HotNets] demonstrate by simulations that Multipath TCP can achieve a better utilization of the available network by using multiple subflows for each Multipath TCP session. Although [RFC6182] assumes that at least one of the communicating hosts has several IP addresses, [HotNets] demonstrates that Multipath TCP is beneficial when both hosts are single-homed. This idea is analysed in more details in [SIGCOMM11] where the Multipath TCP implementation in the Linux kernel is modified to be able to use several subflows from the same IP address. Measurements in a public datacenter show the quantitative benefits of Multipath TCP [SIGCOMM11] in this environment.

Although ECMP is widely used inside datacenters, this is not the only environment where there are different paths between a pair of hosts. ECMP and other load balancing techniques such as Link Aggregation Groups (LAG) are widely used in today's networks and having multiple paths between a pair of single-homed hosts is becoming the norm instead of the exception. Although these multiple paths have often the same cost (from an IGP metrics viewpoint), they do not necessarily have the same performance. For example, [IMC13c] reports the results of a long measurement study showing that load balanced Internet paths between that same pair of hosts can have huge delay differences.

2.2. Cellular/WiFi Offload

A second use case that has been explored by several network researchers is the cellular/WiFi offload use case. Smartphones or other mobile devices equipped with two wireless interfaces are a very common use case for Multipath TCP. In September 2015, this is also the largest deployment of Multipath-TCP enabled devices [IOS7]. It has been briefly discussed during IETF88 [ietf88], but there is no published paper or report that analyses this deployment. For this reason, we only discuss published papers that have mainly used the Multipath TCP implementation in the Linux kernel for their experiments.

The performance of Multipath TCP in wireless networks was briefly evaluated in [NSDI12]. One experiment analyzes the performance of Multipath TCP on a client with two wireless interfaces. This evaluation shows that when the receive window is large, Multipath TCP can efficiently use the two available links. However, if the window becomes smaller, then packets sent on a slow path can block the transmission of packets on a faster path. In some cases, the performance of Multipath TCP over two paths can become lower than the performance of regular TCP over the best performing path. Two heuristics, reinjection and penalization, are proposed in [NSDI12] to solve this identified performance problem. These two heuristics have since been used in the Multipath TCP implementation in the Linux kernel. [CONEXT13] explored the problem in more detail and revealed some other scenarios where Multipath TCP can have difficulties in efficiently pooling the available paths. Improvements to the Multipath TCP implementation in the Linux kernel are proposed in [CONEXT13] to cope with some of these problems.

The first experimental analysis of Multipath TCP in a public wireless environment was presented in [Cellnet12]. These measurements explore the ability of Multipath TCP to use two wireless networks (real WiFi and 3G networks). Three modes of operation are compared. The first mode of operation is the simultaneous use of the two wireless networks. In this mode, Multipath TCP pools the available resources and uses both wireless interfaces. This mode provides fast handover from WiFi to cellular or the opposite when the user moves. Measurements presented in [CACM14] show that the handover from one wireless network to another is not an abrupt process. When a host moves, there are regions where the quality of one of the wireless networks is weaker than the other, but the host considers this wireless network to still be up. When a mobile host enters such regions, its ability to send packets over another wireless network is important to ensure a smooth handover. This is clearly illustrated from the packet trace discussed in [CACM14].

Many cellular networks use volume-based pricing and users often prefer to use unmetered WiFi networks when available instead of metered cellular networks. [Cellnet12] implements support for the MP_PRIO option to explore two other modes of operation.

In the backup mode, Multipath TCP opens a TCP subflow over each interface, but the cellular interface is configured in backup mode. This implies that data only flows over only the WiFi interface when both interfaces are considered to be active. If the WiFi interface fails, then the traffic switches quickly to the cellular interface, ensuring a smooth handover from the user's viewpoint [Cellnet12]. The cost of this approach is that the WiFi and cellular interfaces are likely to remain active all the time since all subflows are established over the two interfaces.

The single-path mode is slightly different. This mode benefits from the break-before-make capability of Multipath TCP. When an MPTCP session is established, a subflow is created over the WiFi interface. No packet is sent over the cellular interface as long as the WiFi interface remains up [Cellnet12]. This implies that the cellular interface can remain idle and battery capacity is preserved. When the WiFi interface fails, a new subflow is established over the cellular interface in order to preserve the established Multipath TCP sessions. Compared to the backup mode described earlier, measurements reported in [Cellnet12] indicate that this mode of operation is characterised by a throughput drop while the cellular interface is brought up and the subflows are reestablished.

From a protocol viewpoint, [Cellnet12] discusses the problem posed by the unreliability of the ADD_ADDR option and proposes a small protocol extension to allow hosts to reliably exchange this option. It would be useful to analyze packet traces to understand whether the unreliability of the REMOVE_ADDR option poses an operational problem in real deployments.

Another study of the performance of Multipath TCP in wireless networks was reported in [IMC13b]. This study uses laptops connected to various cellular ISPs and WiFi hotspots. It compares various file transfer scenarios. [IMC13b] observes that 4-path MPTCP outperforms 2-path MPTCP, especially for larger files. The comparison between LIA, OLIA and Reno does not reveal a significant performance difference for file sizes smaller than 4MB.

A different study of the performance of Multipath TCP with two wireless networks is presented in [INFOCOM14]. In this study the two networks had different qualities : a good network and a lossy network. When using two paths with different packet loss ratios, the Multipath TCP congestion control scheme moves traffic away from the lossy link that is considered to be congested. However, [INFOCOM14] documents an interesting scenario that is summarised in Figure 1.

```
client ----------- path1 -------- server
  |                                 |
  +--------------- path2 ------------+
```

Figure 1: Simple network topology

Initially, the two paths have the same quality and Multipath TCP distributes the load over both of them. During the transfer, the second path becomes lossy, e.g. because the client moves. Multipath TCP detects the packet losses and they are retransmitted over the first path. This enables the data transfer to continue over the first path. However, the subflow over the second path is still up and transmits one packet from time to time. Although the N packets have been acknowledged over the first subflow (at the MPTCP level), they have not been acknowledged at the TCP level over the second subflow. To preserve the continuity of the sequence numbers over the second subflow, TCP will continue to retransmit these segments until either they are acknowledged or the maximum number of retransmissions is reached. This behavior is clearly inefficient and may lead to blocking since the second subflow will consume window space to be able to retransmit these packets. [INFOCOM14] proposes a new Multipath TCP option to solve this problem. In practice, a new TCP option is probably not required. When the client detects that the data transmitted over the second subflow has been acknowledged over the first subflow, it could decide to terminate the second subflow by sending a RST segment. If the interface associated to this subflow is still up, a new subflow could be immediately reestablished. It would then be immediately usable to send new data and would not be forced to first retransmit the previously transmitted data. As of this writing, this dynamic management of the subflows is not yet implemented in the Multipath TCP implementation in the Linux kernel.

2.3. Multipath TCP proxies

As Multipath TCP is not yet widely deployed on both clients and servers, several deployments have used various forms of proxies. Two families of solutions are currently being used or tested [I-D.deng-mptcp-proxy].

A first use case is when a Multipath TCP enabled client wants to use several interfaces to reach a regular TCP server. A typical use case is a smartphone that needs to use both its WiFi and its cellular interface to transfer data. Several types of proxies are possible for this use case. An HTTP proxy deployed on a Multipath TCP capable server would enable the smartphone to use Multipath TCP to access regular web servers. Obviously, this solution only works for applications that rely on HTTP. Another possibility is to use a proxy that can convert any Multipath TCP connection into a regular TCP connection. Multipath TCP-specific proxies have been proposed [I-D.wei-mptcp-proxy-mechanism] [HotMiddlebox13b] [I-D.hampel-mptcp-proxies-anchors].

Another possibility leverages the SOCKS protocol [RFC1928]. SOCKS is often used in enterprise networks to allow clients to reach external servers. For this, the client opens a TCP connection to the SOCKS server that relays it to the final destination. If both the client and the SOCKS server use Multipath TCP, but not the final destination, then Multipath TCP can still be used on the path between the client and the SOCKS server. At IETF'93, Korea Telecom announced that they have deployed in June 2015 a commercial service that uses Multipath TCP on smartphones. These smartphones access regular TCP servers through a SOCKS proxy. This enables them to achieve throughputs of up to 850 Mbps [KT].

Measurements performed with Android smartphones [Mobicom15] show that popular applications work correctly through a SOCKS proxy and Multipath TCP enabled smartphones. Thanks to Multipath TCP, long-lived connections can be spread over the two available interfaces. However, for short-lived connections, most of the data is sent over the initial subflow that is created over the interface corresponding to the default route and the second subflow is almost not used.

A second use case is when Multipath TCP is used by middleboxes, typically inside access networks. Various network operators are discussing and evaluating solutions for hybrid access networks [BBF-WT348]. Such networks arise when a network operator controls two different access network technologies, e.g. wired and cellular, and wants to combine them to improve the bandwidth offered to the endusers [I-D.lhwxz-hybrid-access-network-architecture]. Several solutions are currently investigated for such networks [BBF-WT348]. Figure 2 shows the organisation of such a network. When a client creates a normal TCP connection, it is intercepted by the Hybrid CPE (HPCE) that converts it in a Multipath TCP connection so that it can use the available access networks (DSL and LTE in the example). The Hybrid Access Gateway (HAG) does the opposite to ensure that the regular server sees a normal TCP connection. Some of the solutions that are currently discussed for hybrid networks use Multipath TCP on the HCPE and the HAG. Other solutions rely on tunnels between the HCPE and the HAG [I-D.lhwxz-gre-notifications-hybrid-access].

```
   client --- HCPE ------ DSL ------- HAG --- internet --- server
               |                       |
               +------- LTE -----------+
```

Figure 2: Hybrid Access Network

3. Operational Experience

3.1. Middlebox interference

The interference caused by various types of middleboxes has been an important concern during the design of the Multipath TCP protocol. Three studies on the interactions between Multipath TCP and middleboxes are worth discussing.

The first analysis appears in [IMC11]. This paper was the main motivation for Multipath TCP incorporating various techniques to cope with middlebox interference. More specifically, Multipath TCP has been designed to cope with middleboxes that :

- change source or destination addresses

- change source or destination port numbers

- change TCP sequence numbers

- split or coalesce segments

- remove TCP options

- modify the payload of TCP segments

These middlebox interferences have all been included in the MBtest suite [MBTest]. This test suite is used in [HotMiddlebox13] to verify the reaction of the Multipath TCP implementation in the Linux kernel when faced with middlebox interference. The test environment used for this evaluation is a dual-homed client connected to a single-homed server. The middlebox behavior can be activated on any of the paths. The main results of this analysis are :

- the Multipath TCP implementation in the Linux kernel is not affected by a middlebox that performs NAT or modifies TCP sequence numbers

- when a middlebox removes the MP_CAPABLE option from the initial SYN segment, the Multipath TCP implementation in the Linux kernel falls back correctly to regular TCP

- when a middlebox removes the DSS option from all data segments, the Multipath TCP implementation in the Linux kernel falls back correctly to regular TCP

- when a middlebox performs segment coalescing, the Multipath TCP implementation in the Linux kernel is still able to accurately extract the data corresponding to the indicated mapping

- when a middlebox performs segment splitting, the Multipath TCP implementation in the Linux kernel correctly reassembles the data corresponding to the indicated mapping. [HotMiddlebox13] shows on figure 4 in section 3.3 a corner case with segment splitting that may lead to a desynchronisation between the two hosts.

The interactions between Multipath TCP and real deployed middleboxes is also analyzed in [HotMiddlebox13] and a particular scenario with the FTP application level gateway running on a NAT is described.

Middlebox interference can also be detected by analysing packet traces on Multipath TCP enabled servers. A closer look at the packets received on the multipath-tcp.org server [TMA2015] shows that among the 184,000 Multipath TCP connections, only 125 of them were falling back to regular TCP. These connections originated from 28 different client IP addresses. These include 91 HTTP connections and 34 FTP connections. The FTP interference is expected and due to Application Level Gateways running home routers. The HTTP interference appeared only on the direction from server to client and could have been caused by transparent proxies deployed in cellular or enterprise networks.

From an operational viewpoint, knowing that Multipath TCP can cope with various types of middlebox interference is important. However, there are situations where the network operators need to gather information about where a particular middlebox interference occurs. The tracebox software [tracebox] described in [IMC13a] is an extension of the popular traceroute software that enables network operators to check at which hop a particular field of the TCP header (including options) is modified. It has been used by several network operators to debug various middlebox interference problems. tracebox includes a scripting language that enables its user to specify precisely which packet (including IP and TCP options) is sent by the source. tracebox sends packets with an increasing TTL/HopLimit and compares the information returned in the ICMP messages with the packet that it sent. This enables tracebox to detect any interference caused by middleboxes on a given path. tracebox works better when routers implement the ICMP extension defined in [RFC1812].

Users of the Multipath TCP implementation have reported some experience with middlebox interference. The strangest scenario has been a middlebox that accepts the Multipath TCP options in the SYN segment but later replaces Multipath TCP options with a TCP EOL option [StrangeMbox]. This causes Multipath TCP to perform a fallback to regular TCP without any impact on the application.

3.2. Congestion control

Congestion control has been an important problem for Multipath TCP. The standardised congestion control scheme for Multipath TCP is defined in [RFC6356] and [NSDI11]. This congestion control scheme has been implemented in the Linux implementation of Multipath TCP. Linux uses a modular architecture to support various congestion control schemes. This architecture is applicable for both regular TCP and Multipath TCP. While the coupled congestion control scheme defined in [RFC6356] is the default congestion control scheme in the Linux implementation, other congestion control schemes have been added. The second congestion control scheme is OLIA [CONEXT12]. This congestion control scheme is also an adaptation of the NewReno single path congestion control scheme to support multiple paths. Simulations and measurements have shown that it provides some performance benefits compared to the the default congestion control scheme [CONEXT12]. Measurements over a wide range of parameters reported in [CONEXT13] also indicate some benefits with the OLIA congestion control scheme. Recently, a delay-based congestion control scheme has been ported to the Multipath TCP implementation in the Linux kernel. This congestion control scheme has been evaluated by using simulations in [ICNP12]. The fourth congestion control scheme that has been included in the Linux implementation of Multipath TCP is the BALIA scheme [I-D.walid-mptcp-congestion-control].

These different congestion control schemes have been compared in several articles. [CONEXT13] and [PaaschPhD] compare these algorithms in an emulated environment. The evaluation showed that the delay-based congestion control scheme is less able to efficiently use the available links than the three other schemes. Reports from some users indicate that they seem to favor OLIA.

3.3. Subflow management

The multipath capability of Multipath TCP comes from the utilisation of one subflow per path. The Multipath TCP architecture [RFC6182] and the protocol specification [RFC6824] define the basic usage of the subflows and the protocol mechanisms that are required to create and terminate them. However, there are no guidelines on how subflows are used during the lifetime of a Multipath TCP session. Most of the published experiments with Multipath TCP have been performed in controlled environments. Still, based on the experience running them and discussions on the mptcp-dev mailing list, interesting lessons have been learned about the management of these subflows.

From a subflow viewpoint, the Multipath TCP protocol is completely symmetrical. Both the clients and the server have the capability to create subflows. However in practice the existing Multipath TCP implementations [I-D.eardley-mptcp-implementations-survey] have opted for a strategy where only the client creates new subflows. The main motivation for this strategy is that often the client resides behind a NAT or a firewall, preventing passive subflow openings on the client. Although there are environments such as datacenters where this problem does not occur, as of this writing, no precise requirement has emerged for allowing the server to create new subflows.

3.4. Implemented subflow managers

The Multipath TCP implementation in the Linux kernel includes several strategies to manage the subflows that compose a Multipath TCP session. The basic subflow manager is the full-mesh. As the name implies, it creates a full-mesh of subflows between the communicating hosts.

The most frequent use case for this subflow manager is a multihomed client connected to a single-homed server. In this case, one subflow is created for each interface on the client. The current implementation of the full-mesh subflow manager is static. The subflows are created immediately after the creation of the initial subflow. If one subflow fails during the lifetime of the Multipath TCP session (e.g. due to excessive retransmissions, or the loss of the corresponding interface), it is not always reestablished. There is ongoing work to enhance the full-mesh path manager to deal with such events.

When the server is multihomed, using the full-mesh subflow manager may lead to a large number of subflows being established. For example, consider a dual-homed client connected to a server with three interfaces. In this case, even if the subflows are only created by the client, 6 subflows will be established. This may be excessive in some environments, in particular when the client and/or the server have a large number of interfaces. A recent draft has proposed a Multipath TCP option to negotiate the maximum number of subflows. However, it should be noted that there have been reports on the mptcp-dev mailing indicating that users rely on Multipath TCP to aggregate more than four different interfaces. Thus, there is a need for supporting many interfaces efficiently.

Creating subflows between multihomed clients and servers may sometimes lead to operational issues as observed by discussions on the mptcp-dev mailing list. In some cases the network operators would like to have a better control on how the subflows are created by Multipath TCP [I-D.boucadair-mptcp-max-subflow]. This might require the definition of policy rules to control the operation of the subflow manager. The two scenarios below illustrate some of these requirements.

```
        host1 ----------  switch1 ----- host2
          |                   |            |
          +--------------  switch2 --------+
```

Figure 3: Simple switched network topology

Consider the simple network topology shown in Figure 3. From an operational viewpoint, a network operator could want to create two subflows between the communicating hosts. From a bandwidth utilization viewpoint, the most natural paths are host1-switch1-host2 and host1-switch2-host2. However, a Multipath TCP implementation running on these two hosts may sometimes have difficulties to obtain this result.

To understand the difficulty, let us consider different allocation strategies for the IP addresses. A first strategy is to assign two subnets : subnetA (resp. subnetB) contains the IP addresses of host1’s interface to switch1 (resp. switch2) and host2’s interface to switch1 (resp. switch2). In this case, a Multipath TCP subflow manager should only create one subflow per subnet. To enforce the utilization of these paths, the network operator would have to specify a policy that prefers the subflows in the same subnet over subflows between addresses in different subnets. It should be noted that the policy should probably also specify how the subflow manager should react when an interface or subflow fails.

A second strategy is to use a single subnet for all IP addresses. In this case, it becomes more difficult to specify a policy that indicates which subflows should be established.

The second subflow manager that is currently supported by the Multipath TCP implementation in the Linux kernel is the ndiffport subflow manager. This manager was initially created to exploit the path diversity that exists between single-homed hosts due to the utilization of flow-based load balancing techniques [SIGCOMM11]. This subflow manager creates N subflows between the same pair of IP addresses. The N subflows are created by the client and differ only in the source port selected by the client. It was not designed to be used on multihomed hosts.

3.5. Subflow destination port

The Multipath TCP protocol relies on the token contained in the MP_JOIN option to associate a subflow to an existing Multipath TCP session. This implies that there is no restriction on the source address, destination address and source or destination ports used for the new subflow. The ability to use different source and destination addresses is key to support multihomed servers and clients. The ability to use different destination port numbers is worth discussing because it has operational implications.

For illustration, consider a dual-homed client that creates a second subflow to reach a single-homed server as illustrated in Figure 4.

```
        client ------- r1 --- internet --- server
            |                   |
            +----------r2-------+
```

Figure 4: Multihomed-client connected to single-homed server

When the Multipath TCP implementation in the Linux kernel creates the second subflow it uses the same destination port as the initial subflow. This choice is motivated by the fact that the server might be protected by a firewall and only accept TCP connections (including subflows) on the official port number. Using the same destination port for all subflows is also useful for operators that rely on the port numbers to track application usage in their network.

There have been suggestions from Multipath TCP users to modify the implementation to allow the client to use different destination ports to reach the server. This suggestion seems mainly motivated by traffic shaping middleboxes that are used in some wireless networks. In networks where different shaping rates are associated to different destination port numbers, this could allow Multipath TCP to reach a higher performance. As of this writing, we are not aware of any implementation of this kind of tweaking.

However, from an implementation point-of-view supporting different destination ports for the same Multipath TCP connection can cause some issues. A legacy implementation of a TCP stack creates a listening socket to react upon incoming SYN segments. The listening socket is handling the SYN segments that are sent on a specific port number. Demultiplexing incoming segments can thus be done solely by looking at the IP addresses and the port numbers. With Multipath TCP however, incoming SYN segments may have an MP_JOIN option with a different destination port. This means, that all incoming segments that did not match on an existing listening-socket or an already established socket must be parsed for an eventual MP_JOIN option. This imposes an additional cost on servers, previously not existent on legacy TCP implementations.

3.6. Closing subflows

```
                    client                       server
                       |                           |
   MPTCP: established  |                           | MPTCP: established
   Sub: established    |                           | Sub: established
                       |                           |
                       |         DATA_FIN          |
   MPTCP: close-wait   | <------------------------ | close()   (step 1)
   Sub: established    |         DATA_ACK          |
                       | ------------------------> | MPTCP: fin-wait-2
                       |                           | Sub: established
                       |                           |
                       |  DATA_FIN + subflow-FIN   |
   close()/shutdown()  | ------------------------> | MPTCP: time-wait
   (step 2)            |        DATA_ACK           | Sub: close-wait
   MPTCP: closed       | <------------------------ |
   Sub: fin-wait-2     |                           |
                       |                           |
                       |        subflow-FIN        |
   MPTCP: closed       | <------------------------ | subflow-close()
   Sub: time-wait      |        subflow-ACK        |
   (step 3)            | ------------------------> | MPTCP: time-wait
                       |                           | Sub: closed
                       |                           |
```

Figure 5: Multipath TCP may not be able to avoid time-wait state (even if enforced by the application).

Figure 5 shows a very particular issue within Multipath TCP. Many high-performance applications try to avoid Time-Wait state by deferring the closure of the connection until the peer has sent a FIN. That way, the client on the left of Figure 5 does a passive closure of the connection, transitioning from Close-Wait to Last-ACK and finally freeing the resources after reception of the ACK of the FIN. An application running on top of a Multipath TCP enabled Linux kernel might also use this approach. The difference here is that the close() of the connection (Step 1 in Figure 5) only triggers the sending of a DATA_FIN. Nothing guarantees that the kernel is ready to combine the DATA_FIN with a subflow-FIN. The reception of the

DATA_FIN will make the application trigger the closure of the connection (step 2), trying to avoid Time-Wait state with this late closure. This time, the kernel might decide to combine the DATA_FIN with a subflow-FIN. This decision will be fatal, as the subflow's state machine will not transition from Close-Wait to Last-Ack, but rather go through Fin-Wait-2 into Time-Wait state. The Time-Wait state will consume resources on the host for at least 2 MSL (Maximum Segment Lifetime). Thus, a smart application that tries to avoid Time-Wait state by doing late closure of the connection actually ends up with one of its subflows in Time-Wait state. A high-performance Multipath TCP kernel implementation should honor the desire of the application to do passive closure of the connection and successfully avoid Time-Wait state - even on the subflows.

The solution to this problem lies in an optimistic assumption that a host doing active-closure of a Multipath TCP connection by sending a DATA_FIN will soon also send a FIN on all its subflows. Thus, the passive closer of the connection can simply wait for the peer to send exactly this FIN - enforcing passive closure even on the subflows. Of course, to avoid consuming resources indefinitely, a timer must limit the time our implementation waits for the FIN.

3.7. Packet schedulers

In a Multipath TCP implementation, the packet scheduler is the algorithm that is executed when transmitting each packet to decide on which subflow it needs to be transmitted. The packet scheduler itself does not have any impact on the interoperability of Multipath TCP implementations. However, it may clearly impact the performance of Multipath TCP sessions. The Multipath TCP implementation in the Linux kernel supports a pluggable architecture for the packet scheduler [PaaschPhD]. As of this writing, two schedulers have been implemented: round-robin and lowest-rtt-first. The second scheduler relies on the round-trip-time (rtt) measured on each TCP subflow and sends first segments over the subflow having the lowest round-trip-time. They are compared in [CSWS14]. The experiments and measurements described in [CSWS14] show that the lowest-rtt-first scheduler appears to be the best compromise from a performance viewpoint. Another study of the packet schedulers is presented in [PAMS2014]. This study relies on simulations with the Multipath TCP implementation in the Linux kernel. They compare the lowest-rtt-first with the round-robin and a random scheduler. They show some situations where the lowest-rtt-first scheduler does not perform as well as the other schedulers, but there are many scenarios where the opposite is true. [PAMS2014] notes that "it is highly likely that the optimal scheduling strategy depends on the characteristics of the paths being used."

3.8. Segment size selection

When an application performs a write/send system call, the kernel allocates a packet buffer (sk_buff in Linux) to store the data the application wants to send. The kernel will store at most one MSS (Maximum Segment Size) of data per buffer. As the MSS can differ amongst subflows, an MPTCP implementation must select carefully the MSS used to generate application data. The Linux kernel implementation had various ways of selecting the MSS: minimum or maximum amongst the different subflows. However, these heuristics of MSS selection can cause significant performance issues in some environment. Consider the following example. An MPTCP connection has two established subflows that respectively use a MSS of 1420 and 1428 bytes. If MPTCP selects the maximum, then the application will generate segments of 1428 bytes of data. An MPTCP implementation will have to split the segment in two (a 1420-byte and 8-byte segments) when pushing on the subflow with the smallest MSS. The latter segment will introduce a large overhead as for a single data segment 2 slots will be used in the congestion window (in packets) therefore reducing by roughly twice the potential throughput (in bytes/s) of this subflow. Taking the smallest MSS does not solve the issue as there might be a case where the subflow with the smallest MSS only sends a few packets therefore reducing the potential throughput of the other subflows.

The Linux implementation recently took another approach [DetalMSS]. Instead of selecting the minimum and maximum values, it now dynamically adapts the MSS based on the contribution of all the subflows to the connection’s throughput. For this it computes, for each subflow, the potential throughput achieved by selecting each MSS value and by taking into account the lost space in the cwnd. It then selects the MSS that allows to achieve the highest potential throughput.

3.9. Interactions with the Domain Name System

Multihomed clients such as smartphones can send DNS queries over any of their interfaces. When a single-homed client performs a DNS query, it receives from its local resolver the best answer for its request. If the client is multihomed, the answer returned to the DNS query may vary with the interface over which it has been sent.

```
                      cdn1
                       |
           client -- cellular -- internet -- cdn3
              |                   |
              +----- wifi --------+
                       |
                     cdn2
```

Figure 6: Simple network topology

If the client sends a DNS query over the WiFi interface, the answer will point to the cdn2 server while the same request sent over the cellular interface will point to the cdn1 server. This might cause problems for CDN providers that locate their servers inside ISP networks and have contracts that specify that the CDN server will only be accessed from within this particular ISP. Assume now that both the client and the CDN servers support Multipath TCP. In this case, a Multipath TCP session from cdn1 or cdn2 would potentially use both the cellular network and the WiFi network. Serving the client from cdn2 over the cellular interface could violate the contract between the CDN provider and the network operators. A similar problem occurs with regular TCP if the client caches DNS replies. For example the client obtains a DNS answer over the cellular interface and then stops this interface and starts to use its WiFi interface. If the client retrieves data from cdn1 over its WiFi interface, this may also violate the contract between the CDN and the network operators.

A possible solution to prevent this problem would be to modify the DNS resolution on the client. The client subnet EDNS extension defined in [I-D.ietf-dnsop-edns-client-subnet] could be used for this purpose. When the client sends a DNS query from its WiFi interface, it should also send the client subnet corresponding to the cellular interface in this request. This would indicate to the resolver that the answer should be valid for both the WiFi and the cellular interfaces (e.g., the cdn3 server).

3.10. Captive portals

Multipath TCP enables a host to use different interfaces to reach a server. In theory, this should ensure connectivity when at least one of the interfaces is active. In practice however, there are some particular scenarios with captive portals that may cause operational problems. The reference environment is shown in Figure 7.

```
           client -----  network1
                |
                +------- internet ------------- server
```

Figure 7: Issue with captive portal

The client is attached to two networks : network1 that provides limited connectivity and the entire Internet through the second network interface. In practice, this scenario corresponds to an open WiFi network with a captive portal for network1 and a cellular service for the second interface. On many smartphones, the WiFi interface is preferred over the cellular interface. If the smartphone learns a default route via both interfaces, it will typically prefer to use the WiFi interface to send its DNS request and create the first subflow. This is not optimal with Multipath TCP. A better approach would probably be to try a few attempts on the WiFi interface and then try to use the second interface for the initial subflow as well.

3.11. Stateless webservers

MPTCP has been designed to interoperate with webservers that benefit from SYN-cookies to protect against SYN-flooding attacks [RFC4987]. MPTCP achieves this by echoing the keys negotiated during the MP_CAPABLE handshake in the third ACK of the 3-way handshake. Reception of this third ACK then allows the server to reconstruct the state specific to MPTCP.

However, one caveat to this mechanism is the non-reliable nature of the third ACK. Indeed, when the third ACK gets lost, the server will not be able to reconstruct the MPTCP-state. MPTCP will fallback to regular TCP in this case. This is in contrast to regular TCP, as clients usually start the application’s transaction by sending data to the server. This data-segment (that is sent reliably by TCP) enables stateless servers to create the TCP-related state, even in case the third ACK has been lost.

This issue might be considered as a minor one for MPTCP. Losing the third ACK should only happen when packet loss is high. However, when packet-loss is high MPTCP provides a lot of benefits as it can move traffic away from the lossy link. It is undesirable that MPTCP has a higher chance to fall back to regular TCP in those lossy environments.

[I-D.paasch-mptcp-syncookies] discusses this issue and suggests a modified handshake mechanism that ensures reliable delivery of the MP_CAPABLE, following the 3-way handshake. This modification will make MPTCP reliable, even in lossy environments when servers need to use SYN-cookies to protect against SYN-flooding attacks.

3.12. Loadbalanced serverfarms

Large-scale serverfarms typically deploy thousands of servers behind a single virtual IP (VIP). Steering traffic to these servers is done through layer-4 loadbalancers that ensure that a TCP-flow will always be routed to the same server [Presto08].

As Multipath TCP uses multiple different TCP subflows to steer the traffic across the different paths, loadbalancers need to ensure that all these subflows are routed to the same server. This implies that the loadbalancers need to track the MPTCP-related state, allowing them to parse the token in the MP_JOIN and assign those subflows to the appropriate server. However, serverfarms typically deploy multiple of these loadbalancers for reliability and capacity reasons. As a TCP subflow might get routed to any of these loadbalancers, they would need to synchronize the MPTCP-related state - a solution that is not feasible at large scale.

The token (carried in the MP_JOIN) contains the information indicating which MPTCP-session the subflow belongs to. As the token is a hash of the key, servers are not able to generate the token in such a way that the token can provide the necessary information to the loadbalancers which would allow them to route TCP subflows to the appropriate server. [I-D.paasch-mptcp-loadbalancer] discusses this issue in detail and suggests two alternative MP_CAPABLE handshakes to overcome these. As of September 2015, it is not yet clear how MPTCP might accomodate such use-case to enable its deployment within loadbalanced serverfarms.

4. Conclusion

In this document, we have documented a few years of experience with Multipath TCP. The information presented in this document was gathered from scientific publications and discussions with various users of the Multipath TCP implementation in the Linux kernel.

5. Acknowledgements

This work was partially supported by the FP7-Trilogy2 project. We would like to thank all the implementers and users of the Multipath TCP implementation in the Linux kernel. This document has benefited from the comments of John Ronan, Yoshifumi Nishida, Phil Eardley and Jaehyun Hwang.

6. Informative References

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

[BBF-WT348]
Fabregas (Ed), G., "WT-348 - Hybrid Access for Broadband Networks", Broadband Forum, contribution bbf2014.1139.04 , June 2015.

[CACM14] Paasch, C. and O. Bonaventure, "Multipath TCP", Communications of the ACM, 57(4):51-57 , April 2014, <http://inl.info.ucl.ac.be/publications/multipath-tcp>.

[CONEXT12]
Khalili, R., Gast, N., Popovic, M., Upadhyay, U., and J. Leboudec, "MPTCP is not pareto-optimal performance issues and a possible solution", Proceedings of the 8th international conference on Emerging networking experiments and technologies (CoNEXT12) , 2012.

[CONEXT13]
Paasch, C., Khalili, R., and O. Bonaventure, "On the Benefits of Applying Experimental Design to Improve Multipath TCP", Conference on emerging Networking EXperiments and Technologies (CoNEXT) , December 2013, <http://inl.info.ucl.ac.be/publications/benefits-applying-experimental-design-improve-multipath-tcp>.

[CSWS14] Paasch, C., Ferlin, S., Alay, O., and O. Bonaventure, "Experimental Evaluation of Multipath TCP Schedulers", SIGCOMM CSWS2014 workshop , August 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>.

[DetalMSS]
Detal, G., "Adaptive MSS value", Post on the mptcp-dev mailing list , September 2014, <https://listes-2.sipr.ucl.ac.be/sympa/arc/mptcp-dev/2014-09/msg00130.html>.

[FreeBSD-MPTCP]
Williams, N., "Multipath TCP For FreeBSD Kernel Patch v0.5", n.d., <http://caia.swin.edu.au/urp/newtcp/mptcp>.

[HotMiddlebox13]
Hesmans, B., Duchene, F., Paasch, C., Detal, G., and O. Bonaventure, "Are TCP Extensions Middlebox-proof?", CoNEXT workshop HotMiddlebox , December 2013, <http://inl.info.ucl.ac.be/publications/are-tcp-extensions-middlebox-proof>.

[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>.

[HotNets] Raiciu, C., Pluntke, C., Barre, S., Greenhalgh, A., Wischik, D., and M. Handley, "Data center networking with multipath TCP", Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets-IX) , 2010, <http://doi.acm.org/10.1145/1868447.1868457>.

[I-D.boucadair-mptcp-max-subflow]
Boucadair, M. and C. Jacquenet, "Negotiating the Maximum Number of MPTCP Subflows", draft-boucadair-mptcp-max-subflow-00 (work in progress), June 2015.

[I-D.deng-mptcp-proxy]
Lingli, D., Liu, D., Sun, T., Boucadair, M., and G. Cauchie, "Use-cases and Requirements for MPTCP Proxy in ISP Networks", draft-deng-mptcp-proxy-01 (work in progress), October 2014.

[I-D.eardley-mptcp-implementations-survey]
Eardley, P., "Survey of MPTCP Implementations", draft-eardley-mptcp-implementations-survey-02 (work in progress), July 2013.

[I-D.hampel-mptcp-proxies-anchors]
Hampel, G. and T. Klein, "MPTCP Proxies and Anchors", draft-hampel-mptcp-proxies-anchors-00 (work in progress), February 2012.

[I-D.ietf-dnsop-edns-client-subnet]
Contavalli, C., Gaast, W., Lawrence, D., and W. Kumari, "Client Subnet in DNS Queries", draft-ietf-dnsop-edns-client-subnet-04 (work in progress), September 2015.

[I-D.lhwxz-gre-notifications-hybrid-access]
Leymann, N., Heidemann, C., Wasserman, M., Xue, L., and M. Zhang, "GRE Notifications for Hybrid Access", draft-lhwxz-gre-notifications-hybrid-access-01 (work in progress), January 2015.

[I-D.lhwxz-hybrid-access-network-architecture]
Leymann, N., Heidemann, C., Wasserman, M., Xue, L., and M. Zhang, "Hybrid Access Network Architecture", draft-lhwxz-hybrid-access-network-architecture-02 (work in progress), January 2015.

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

[I-D.paasch-mptcp-syncookies]
Paasch, C., Biswas, A., and D. Haas, "Making Multipath TCP robust for stateless webservers", draft-paasch-mptcp-syncookies-02 (work in progress), October 2015.

[I-D.walid-mptcp-congestion-control]
Walid, A., Peng, Q., Hwang, J., and S. Low, "Balanced Linked Adaptation Congestion Control Algorithm for MPTCP", draft-walid-mptcp-congestion-control-03 (work in progress), July 2015.

[I-D.wei-mptcp-proxy-mechanism]
Wei, X., Xiong, C., and E. Ed, "MPTCP proxy mechanisms", draft-wei-mptcp-proxy-mechanism-02 (work in progress), June 2015.

[ICNP12] Cao, Y., Xu, M., and X. Fu, "Delay-based congestion control for multipath TCP", 20th IEEE International Conference on Network Protocols (ICNP) , 2012.

[IMC11] Honda, M., Nishida, Y., Raiciu, C., Greenhalgh, A., Handley, M., and H. Tokuda, "Is it still possible to extend TCP?", Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference (IMC ’11) , 2011, <http://doi.acm.org/10.1145/2068816.2068834>.

[IMC13a] Detal, G., Hesmans, B., Bonaventure, O., Vanaubel, Y., and B. Donnet, "Revealing Middlebox Interference with Tracebox", Proceedings of the 2013 ACM SIGCOMM conference on Internet measurement conference , 2013, <http://inl.info.ucl.ac.be/publications/revealing-middlebox-interference-tracebox>.

[IMC13b] Chen, Y., Lim, Y., Gibbens, R., Nahum, E., Khalili, R., and D. Towsley, "A measurement-based study of MultiPath TCP performance over wireless network", Proceedings of the 2013 conference on Internet measurement conference (IMC ’13) , n.d., <http://doi.acm.org/10.1145/2504730.2504751>.

[IMC13c] Pelsser, C., Cittadini, L., Vissicchio, S., and R. Bush, "From Paris to Tokyo on the suitability of ping to measure latency", Proceedings of the 2013 conference on Internet measurement conference (IMC ’13) , 2013, <http://doi.acm.org/10.1145/2504730.2504765>.

[INFOCOM14]
Lim, Y., Chen, Y., Nahum, E., Towsley, D., and K. Lee, "Cross-Layer Path Management in Multi-path Transport Protocol for Mobile Devices", IEEE INFOCOM’14 , 2014.

[IOS7] "Multipath TCP Support in iOS 7", January 2014, <http://support.apple.com/kb/HT5977>.

[KT] Seo, S., "KT’s GiGA LTE", July 2015, <https://www.ietf.org/proceedings/93/slides/slides-93-mptcp-3.pdf>.

[MBTest] Hesmans, B., "MBTest", 2013, <https://bitbucket.org/bhesmans/mbtest>.

[MPTCPBIB]
Bonaventure, O., "Multipath TCP - An annotated bibliography", Technical report , April 2015, <https://github.com/obonaventure/mptcp-bib>.

[Mobicom15]
De Coninck, Q., Baerts, M., Hesmans, B., and O. Bonaventure, "Poster - Evaluating Android Applications with Multipath TCP", Mobicom 2015 (Poster) , September 2015.

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

[NSDI11] Wischik, D., Raiciu, C., Greenhalgh, A., and M. Handley, "Design, implementation and evaluation of congestion control for Multipath TCP", In Proceedings of the 8th USENIX conference on Networked systems design and implementation (NSDI11) , 2011.

[NSDI12] 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 of Networked Systems Design and Implementation (NSDI12) , April 2012, <http://inl.info.ucl.ac.be/publications/how-hard-can-it-be-designing-and-implementing-deployable-multipath-tcp>.

[PAMS2014]
Arzani, B., Gurney, A., Cheng, S., Guerin, R., and B. Loo, "Impact of Path Selection and Scheduling Policies on MPTCP Performance", PAMS2014 , 2014.

[PaaschPhD]
Paasch, C., "Improving Multipath TCP", Ph.D. Thesis , November 2014, <http://inl.info.ucl.ac.be/publications/improving-multipath-tcp>.

[Presto08]
Greenberg, A., Lahiri, P., Maltz, D., Parveen, P., and S. Sengupta, "Towards a Next Generation Data Center Architecture - Scalability and Commoditization", ACM PRESTO 2008 , August 2008, <http://dl.acm.org/citation.cfm?id=1397732>.

[RFC1812] Baker, F., Ed., "Requirements for IP Version 4 Routers", RFC 1812, DOI 10.17487/RFC1812, June 1995, <http://www.rfc-editor.org/info/rfc1812>.

[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>.

[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>.

[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>.

[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>.

[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>.

[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 , n.d., <http://doi.acm.org/10.1145/2018436.2018467>.

[StrangeMbox] Bonaventure, O., "Multipath TCP through a strange middlebox", Blog post , January 2015, <http://blog.multipath-tcp.org/blog/html/2015/01/30/multipath_tcp_through_a_strange_middlebox.html>.

[TMA2015] Hesmans, B., Tran Viet, H., Sadre, R., and O. Bonaventure, "A First Look at Real Multipath TCP Traffic", Traffic Monitoring and Analysis , 2015, <http://inl.info.ucl.ac.be/publications/first-look-real-multipath-tcp-traffic>.

[ietf88] Stewart, L., "IETF’88 Meeting minutes of the MPTCP working group", n.d., <http://tools.ietf.org/wg/mptcp/minutes?item=minutes-88-mptcp.html>.

[tracebox] Detal, G. and O. Tilmans, "tracebox", 2013, <http://www.tracebox.org>.

Appendix A. Changelog

This section should be removed before final publication

- o initial version : September 16th, 2014 : Added section Section 3.8 that discusses some performance problems that appeared with the

Linux implementation when using subflows having different MSS values

- update with a description of the middlebox that replaces an unknown TCP option with EOL [StrangeMbox]
- version ietf-02 : July 2015, answer to last call comments
  - Reorganised text to better separate use cases and operational experience
  - New use case on Multipath TCP proxies in Section 2.3
  - Added some text on middleboxes in Section 3.1
  - Removed the discussion on SDN
  - Restructured text and improved writing in some parts
- version ietf-03 : September 2015, answer to comments from Phil Eardley
  - Improved introduction
  - Added details about using SOCKS and Korea Telecom's use-case in Section 2.3.
  - Added issue around clients caching DNS-results in Section 3.9
  - Explained issue of MPTCP with stateless webservers Section 3.11
  - Added description of MPTCP's use behind layer-4 loadbalancers Section 3.12
  - Restructured text and improved writing in some parts

Authors' Addresses

Olivier Bonaventure
UCLouvain

Email: Olivier.Bonaventure@uclouvain.be

Christoph Paasch
Apple, Inc.

Email: cpaasch@apple.com

Gregory Detal
UCLouvain and Tessares

Email: Gregory.Detal@tessares.net

MPTCP Working Group
Internet-Draft
Intended status: Informational
Expires: April 30, 2017

O. Bonaventure
UCLouvain
C. Paasch
Apple, Inc.
G. Detal
Tessares

October 27, 2016

Use Cases and Operational Experience with Multipath TCP
draft-ietf-mptcp-experience-07

Abstract

This document discusses both use cases and operational experience with Multipath TCP in real networks. It lists several prominent use cases where Multipath TCP has been considered and is being used. It also gives insight to some heuristics and decisions that have helped to realize these use cases and suggests possible improvements.

Status of this Memo

This Internet-Draft will expire on April 30, 2017.

Copyright Notice

Table of Contents

1. Introduction

Multipath TCP was specified in [RFC6824] and five independent implementations have been developed. As of November 2016, Multipath TCP has been or is being implemented on the following platforms:

- Linux kernel [MultipathTCP-Linux]
- Apple iOS and macOS [Apple-MPTCP]
- Citrix load balancers
- FreeBSD [FreeBSD-MPTCP]
- Oracle Solaris

The first three implementations are known to interoperate. Three of these implementations are open-source (Linux kernel, FreeBSD and Apple’s iOS and macOS). Apple’s implementation is widely deployed.

Since the publication of [RFC6824] as an experimental RFC, experience has been gathered by various network researchers and users about the operational issues that arise when Multipath TCP is used in today’s Internet.

When the MPTCP working group was created, several use cases for Multipath TCP were identified [RFC6182]. Since then, other use cases have been proposed and some have been tested and even deployed. We describe these use cases in Section 2.

Section 3 focuses on the operational experience with Multipath TCP. Most of this experience comes from the utilization of the Multipath TCP implementation in the Linux kernel [MultipathTCP-Linux]. This open-source implementation has been downloaded and is used by thousands of users all over the world. Many of these users have provided direct or indirect feedback by writing documents (scientific articles or blog messages) or posting to the mptcp-dev mailing list (see https://listes-2.sipr.ucl.ac.be/sympa/arc/mptcp-dev). This Multipath TCP implementation is actively maintained and continuously improved. It is used on various types of hosts, ranging from smartphones or embedded routers to high-end servers.

The Multipath TCP implementation in the Linux kernel is not, by far, the most widespread deployment of Multipath TCP. Since September 2013, Multipath TCP is also supported on smartphones and tablets since iOS7 [IOS7]. There are likely hundreds of millions of Multipath TCP enabled devices. This Multipath TCP implementation is currently only used to support the Siri voice recognition/control application. Some lessons learned from this deployment are described in [IETFJ].

Section 3 is organized as follows. Supporting the middleboxes was one of the difficult issues in designing the Multipath TCP protocol. We explain in Section 3.1 which types of middleboxes the Linux Kernel implementation of Multipath TCP supports and how it reacts upon encountering these. Section 3.2 summarizes the MPTCP specific congestion controls that have been implemented. Section 3.3 to Section 3.7 discuss heuristics and issues with respect to subflow management as well as the scheduling across the subflows. Section 3.8 explains some problems that occurred with subflows having different Maximum Segment Size (MSS) values. Section 3.9 presents issues with respect to content delivery networks and suggests a solution to this issue. Finally, Section 3.10 documents an issue with captive portals where MPTCP will behave sub optimally.

2. Use cases

Multipath TCP has been tested in several use cases. There is already an abundant scientific literature on Multipath TCP [MPTCPBIB]. Several of the papers published in the scientific literature have identified possible improvements that are worth being discussed here.

2.1. Datacenters

A first, although initially unexpected, documented use case for Multipath TCP has been in datacenters [HotNets][SIGCOMM11]. Today's datacenters are designed to provide several paths between single-homed servers. The multiplicity of these paths comes from the utilization of Equal Cost Multipath (ECMP) and other load balancing techniques inside the datacenter. Most of the deployed load balancing techniques in datacenters rely on hashes computed over the five tuple. Thus all packets from the same TCP connection follow the same path and so are not reordered. The results in [HotNets] demonstrate by simulations that Multipath TCP can achieve a better utilization of the available network by using multiple subflows for each Multipath TCP session. Although [RFC6182] assumes that at least one of the communicating hosts has several IP addresses, [HotNets] demonstrates that Multipath TCP is beneficial when both hosts are single-homed. This idea is analyzed in more details in [SIGCOMM11] where the Multipath TCP implementation in the Linux kernel is modified to be able to use several subflows from the same IP address. Measurements in a public datacenter show the quantitative benefits of Multipath TCP [SIGCOMM11] in this environment.

Although ECMP is widely used inside datacenters, this is not the only environment where there are different paths between a pair of hosts. ECMP and other load balancing techniques such as Link Aggregation Groups (LAG) are widely used in today's networks and having multiple paths between a pair of single-homed hosts is becoming the norm instead of the exception. Although these multiple paths have often the same cost (from an IGP metrics viewpoint), they do not necessarily have the same performance. For example, [IMC13c] reports the results of a long measurement study showing that load balanced Internet paths between that same pair of hosts can have huge delay differences.

2.2. Cellular/WiFi Offload

A second use case that has been explored by several network researchers is the cellular/WiFi offload use case. Smartphones or other mobile devices equipped with two wireless interfaces are a very common use case for Multipath TCP. In September 2015, this is also the largest deployment of Multipath-TCP enabled devices [IOS7]. It has been briefly discussed during IETF88 [ietf88], but there is no published paper or report that analyses this deployment. For this reason, we only discuss published papers that have mainly used the Multipath TCP implementation in the Linux kernel for their experiments.

The performance of Multipath TCP in wireless networks was briefly evaluated in [NSDI12]. One experiment analyzes the performance of Multipath TCP on a client with two wireless interfaces. This evaluation shows that when the receive window is large, Multipath TCP can efficiently use the two available links. However, if the window becomes smaller, then packets sent on a slow path can block the transmission of packets on a faster path. In some cases, the performance of Multipath TCP over two paths can become lower than the performance of regular TCP over the best performing path. Two heuristics, reinjection and penalization, are proposed in [NSDI12] to solve this identified performance problem. These two heuristics have since been used in the Multipath TCP implementation in the Linux kernel. [CONEXT13] explored the problem in more detail and revealed some other scenarios where Multipath TCP can have difficulties in efficiently pooling the available paths. Improvements to the Multipath TCP implementation in the Linux kernel are proposed in [CONEXT13] to cope with some of these problems.

The first experimental analysis of Multipath TCP in a public wireless environment was presented in [Cellnet12]. These measurements explore the ability of Multipath TCP to use two wireless networks (real WiFi and 3G networks). Three modes of operation are compared. The first mode of operation is the simultaneous use of the two wireless networks. In this mode, Multipath TCP pools the available resources and uses both wireless interfaces. This mode provides fast handover from WiFi to cellular or the opposite when the user moves. Measurements presented in [CACM14] show that the handover from one wireless network to another is not an abrupt process. When a host moves, there are regions where the quality of one of the wireless networks is weaker than the other, but the host considers this wireless network to still be up. When a mobile host enters such regions, its ability to send packets over another wireless network is important to ensure a smooth handover. This is clearly illustrated from the packet trace discussed in [CACM14].

Many cellular networks use volume-based pricing and users often prefer to use unmetered WiFi networks when available instead of metered cellular networks. [Cellnet12] implements support for the MP_PRIO option to explore two other modes of operation.

In the backup mode, Multipath TCP opens a TCP subflow over each interface, but the cellular interface is configured in backup mode.

This implies that data flows only over the WiFi interface when both interfaces are considered to be active. If the WiFi interface fails, then the traffic switches quickly to the cellular interface, ensuring a smooth handover from the user’s viewpoint [Cellnet12]. The cost of this approach is that the WiFi and cellular interfaces are likely to remain active all the time since all subflows are established over the two interfaces.

The single-path mode is slightly different. This mode benefits from the break-before-make capability of Multipath TCP. When an MPTCP session is established, a subflow is created over the WiFi interface. No packet is sent over the cellular interface as long as the WiFi interface remains up [Cellnet12]. This implies that the cellular interface can remain idle and battery capacity is preserved. When the WiFi interface fails, a new subflow is established over the cellular interface in order to preserve the established Multipath TCP sessions. Compared to the backup mode described earlier, measurements reported in [Cellnet12] indicate that this mode of operation is characterized by a throughput drop while the cellular interface is brought up and the subflows are reestablished.

From a protocol viewpoint, [Cellnet12] discusses the problem posed by the unreliability of the REMOVE_ADDR option and proposes a small protocol extension to allow hosts to reliably exchange this option. It would be useful to analyze packet traces to understand whether the unreliability of the REMOVE_ADDR option poses an operational problem in real deployments.

Another study of the performance of Multipath TCP in wireless networks was reported in [IMC13b]. This study uses laptops connected to various cellular ISPs and WiFi hotspots. It compares various file transfer scenarios. [IMC13b] observes that 4-path MPTCP outperforms 2-path MPTCP, especially for larger files. However, for three congestion control algorithms (LIA, OLIA and Reno - see Section 3.2), there is no significant performance difference for file sizes smaller than 4MB.

A different study of the performance of Multipath TCP with two wireless networks is presented in [INFOCOM14]. In this study the two networks had different qualities : a good network and a lossy network. When using two paths with different packet loss ratios, the Multipath TCP congestion control scheme moves traffic away from the lossy link that is considered to be congested. However, [INFOCOM14] documents an interesting scenario that is summarized hereafter.

```
   client ----------- path1 -------- server
     |                                  |
     +--------------- path2 ------------+
```

Figure 1: Simple network topology

Initially, the two paths in Figure 1 have the same quality and Multipath TCP distributes the load over both of them. During the transfer, the path2 becomes lossy, e.g. because the client moves. Multipath TCP detects the packet losses and they are retransmitted over path1. This enables the data transfer to continue over this path. However, the subflow over path2 is still up and transmits one packet from time to time. Although the N packets have been acknowledged over the first subflow (at the MPTCP level), they have not been acknowledged at the TCP level over the second subflow. To preserve the continuity of the sequence numbers over the second subflow, TCP will continue to retransmit these segments until either they are acknowledged or the maximum number of retransmissions is reached. This behavior is clearly inefficient and may lead to blocking since the second subflow will consume window space to be able to retransmit these packets. [INFOCOM14] proposes a new Multipath TCP option to solve this problem. In practice, a new TCP option is probably not required. When the client detects that the data transmitted over the second subflow has been acknowledged over the first subflow, it could decide to terminate the second subflow by sending a RST segment. If the interface associated to this subflow is still up, a new subflow could be immediately reestablished. It would then be immediately usable to send new data and would not be forced to first retransmit the previously transmitted data. As of this writing, this dynamic management of the subflows is not yet implemented in the Multipath TCP implementation in the Linux kernel.

Some studies have started to analyze the performance of Multipath TCP on smartphones with real applications. In contrast with the bulk transfers that are used by many publications, many deployed applications do not exchange huge amounts of data and mainly use small connections. [COMMAG2016] proposes a software testing framework that allows to automate Android applications to study their interactions with Multipath TCP. [PAM2016] analyses a one-month packet trace of all the packets exchanged by a dozen of smartphones used by regular users. This analysis reveals that short connections are important on smartphones and that the main benefit of using Multipath TCP on smartphones is the ability to perform seamless handovers between different wireless networks. Long connections benefit from these handovers.

2.3. Multipath TCP proxies

As Multipath TCP is not yet widely deployed on both clients and servers, several deployments have used various forms of proxies. Two families of solutions are currently being used or tested.

A first use case is when a Multipath TCP enabled client wants to use several interfaces to reach a regular TCP server. A typical use case is a smartphone that needs to use both its WiFi and its cellular interface to transfer data. Several types of proxies are possible for this use case. An HTTP proxy deployed on a Multipath TCP capable server would enable the smartphone to use Multipath TCP to access regular web servers. Obviously, this solution only works for applications that rely on HTTP. Another possibility is to use a proxy that can convert any Multipath TCP connection into a regular TCP connection. Multipath TCP-specific proxies have been proposed [HotMiddlebox13b] [HAMPEL].

Another possibility leverages the SOCKS protocol [RFC1928]. SOCKS is often used in enterprise networks to allow clients to reach external servers. For this, the client opens a TCP connection to the SOCKS server that relays it to the final destination. If both the client and the SOCKS server use Multipath TCP, but not the final destination, then Multipath TCP can still be used on the path between the clients and the SOCKS server. At IETF’93, Korea Telecom announced that they have deployed in June 2015 a commercial service that uses Multipath TCP on smartphones. These smartphones access regular TCP servers through a SOCKS proxy. This enables them to achieve throughputs of up to 850 Mbps [KT].

Measurements performed with Android smartphones [Mobicom15] show that popular applications work correctly through a SOCKS proxy and Multipath TCP enabled smartphones. Thanks to Multipath TCP, long-lived connections can be spread over the two available interfaces. However, for short-lived connections, most of the data is sent over the initial subflow that is created over the interface corresponding to the default route and the second subflow is almost not used [PAM2016].

A second use case is when Multipath TCP is used by middleboxes, typically inside access networks. Various network operators are discussing and evaluating solutions for hybrid access networks [TR-348]. Such networks arise when a network operator controls two different access network technologies, e.g. wired and cellular, and wants to combine them to improve the bandwidth offered to the endusers [I-D.lhwxz-hybrid-access-network-architecture]. Several solutions are currently investigated for such networks [TR-348]. Figure 2 shows the organization of such a network. When a client creates a normal TCP connection, it is intercepted by the Hybrid CPE (HPCE) that converts it in a Multipath TCP connection so that it can use the available access networks (DSL and LTE in the example). The Hybrid Access Gateway (HAG) does the opposite to ensure that the regular server sees a normal TCP connection. Some of the solutions currently discussed for hybrid networks use Multipath TCP on the HCPE and the HAG. Other solutions rely on tunnels between the HCPE and the HAG [I-D.lhwxz-gre-notifications-hybrid-access].

```
client --- HCPE ------ DSL ------- HAG --- internet --- server
            |                       |
            +------- LTE -----------+
```

Figure 2: Hybrid Access Network

3. Operational Experience

3.1. Middlebox interference

The interference caused by various types of middleboxes has been an important concern during the design of the Multipath TCP protocol. Three studies on the interactions between Multipath TCP and middleboxes are worth discussing.

The first analysis appears in [IMC11]. This paper was the main motivation for Multipath TCP incorporating various techniques to cope with middlebox interference. More specifically, Multipath TCP has been designed to cope with middleboxes that :

- change source or destination addresses
- change source or destination port numbers
- change TCP sequence numbers
- split or coalesce segments
- remove TCP options
- modify the payload of TCP segments

These middlebox interferences have all been included in the MBtest suite [MBTest]. This test suite is used in [HotMiddlebox13] to verify the reaction of the Multipath TCP implementation in the Linux kernel [MultipathTCP-Linux] when faced with middlebox interference. The test environment used for this evaluation is a dual-homed client connected to a single-homed server. The middlebox behavior can be activated on any of the paths. The main results of this analysis are :

- the Multipath TCP implementation in the Linux kernel, is not affected by a middlebox that performs NAT or modifies TCP sequence numbers
- when a middlebox removes the MP_CAPABLE option from the initial SYN segment, the Multipath TCP implementation in the Linux kernel falls back correctly to regular TCP
- when a middlebox removes the DSS option from all data segments, the Multipath TCP implementation in the Linux kernel falls back correctly to regular TCP

- when a middlebox performs segment coalescing, the Multipath TCP implementation in the Linux kernel is still able to accurately extract the data corresponding to the indicated mapping

- when a middlebox performs segment splitting, the Multipath TCP implementation in the Linux kernel correctly reassembles the data corresponding to the indicated mapping. [HotMiddlebox13] shows on figure 4 in section 3.3 a corner case with segment splitting that may lead to a desynchronization between the two hosts.

The interactions between Multipath TCP and real deployed middleboxes is also analyzed in [HotMiddlebox13] and a particular scenario with the FTP application level gateway running on a NAT is described.

Middlebox interference can also be detected by analyzing packet traces on Multipath TCP enabled servers. A closer look at the packets received on the multipath-tcp.org server [TMA2015] shows that among the 184,000 Multipath TCP connections, only 125 of them were falling back to regular TCP. These connections originated from 28 different client IP addresses. These include 91 HTTP connections and 34 FTP connections. The FTP interference is expected since Application Level Gateways used for FTP modify the TCP payload and the DSS Checksum detects these modifications. The HTTP interference appeared only on the direction from server to client and could have been caused by transparent proxies deployed in cellular or enterprise networks. A longer trace is discussed in [COMCOM2016] and similar conclusions about the middlebox interference are provided.

From an operational viewpoint, knowing that Multipath TCP can cope with various types of middlebox interference is important. However, there are situations where the network operators need to gather information about where a particular middlebox interference occurs. The tracebox software [tracebox] described in [IMC13a] is an extension of the popular traceroute software that enables network operators to check at which hop a particular field of the TCP header (including options) is modified. It has been used by several network operators to debug various middlebox interference problems. Experience with tracebox indicates that supporting the ICMP extension defined in [RFC1812] makes it easier to debug middlebox problems in IPv4 networks.

Users of the Multipath TCP implementation have reported some experience with middlebox interference. The strangest scenario has been a middlebox that accepts the Multipath TCP options in the SYN segment but later replaces Multipath TCP options with a TCP EOL option [StrangeMbox]. This causes Multipath TCP to perform a fallback to regular TCP without any impact on the application.

3.2. Congestion control

Congestion control has been an important challenge for Multipath TCP. The congestion control scheme specified for Multipath TCP is defined in [RFC6356]. A detailed description of this algorithm is provided in [NSDI11]. This congestion control scheme has been implemented in the Linux implementation of Multipath TCP. Linux uses a modular architecture to support various congestion control schemes. This architecture is applicable for both regular TCP and Multipath TCP. While the coupled congestion control scheme defined in [RFC6356] is the default congestion control scheme in the Linux implementation, other congestion control schemes have been added. The second congestion control scheme is OLIA [CONEXT12]. This congestion control scheme is also an adaptation of the NewReno single path congestion control scheme to support multiple paths. Simulations and measurements have shown that it provides some performance benefits compared to the default congestion control scheme [CONEXT12]. Measurements over a wide range of parameters reported in [CONEXT13] also indicate some benefits with the OLIA congestion control scheme. Recently, a delay-based congestion control scheme has been ported to the Multipath TCP implementation in the Linux kernel. This congestion control scheme has been evaluated by using simulations in [ICNP12] and measurements in [PaaschPhD]. The fourth congestion control scheme that has been included in the Linux implementation of Multipath TCP is the BALIA scheme that provides a better balance between TCP friendliness, responsiveness, and window oscillation [BALIA].

These different congestion control schemes have been compared in several articles. [CONEXT13] and [PaaschPhD] compare these algorithms in an emulated environment. The evaluation showed that the delay-based congestion control scheme is less able to efficiently use the available links than the three other schemes.

3.3. Subflow management

The multipath capability of Multipath TCP comes from the utilization of one subflow per path. The Multipath TCP architecture [RFC6182] and the protocol specification [RFC6824] define the basic usage of the subflows and the protocol mechanisms that are required to create and terminate them. However, there are no guidelines on how subflows are used during the lifetime of a Multipath TCP session. Most of the published experiments with Multipath TCP have been performed in controlled environments. Still, based on the experience running them and discussions on the mptcp-dev mailing list, interesting lessons have been learned about the management of these subflows.

From a subflow viewpoint, the Multipath TCP protocol is completely symmetrical. Both the clients and the server have the capability to create subflows. However in practice the existing Multipath TCP implementations have opted for a strategy where only the client creates new subflows. The main motivation for this strategy is that often the client resides behind a NAT or a firewall, preventing passive subflow openings on the client. Although there are environments such as datacenters where this problem does not occur, as of this writing, no precise requirement has emerged for allowing the server to create new subflows.

3.4. Implemented subflow managers

The Multipath TCP implementation in the Linux kernel includes several strategies to manage the subflows that compose a Multipath TCP session. The basic subflow manager is the full-mesh. As the name implies, it creates a full-mesh of subflows between the communicating hosts.

The most frequent use case for this subflow manager is a multihomed client connected to a single-homed server. In this case, one subflow is created for each interface on the client. The current implementation of the full-mesh subflow manager is static. The subflows are created immediately after the creation of the initial subflow. If one subflow fails during the lifetime of the Multipath TCP session (e.g. due to excessive retransmissions, or the loss of the corresponding interface), it is not always reestablished. There is ongoing work to enhance the full-mesh path manager to deal with such events.

When the server is multihomed, using the full-mesh subflow manager may lead to a large number of subflows being established. For example, consider a dual-homed client connected to a server with three interfaces. In this case, even if the subflows are only created by the client, 6 subflows will be established. This may be excessive in some environments, in particular when the client and/or the server have a large number of interfaces. Implementations should limit the number of subflows that are used.

Creating subflows between multihomed clients and servers may sometimes lead to operational issues as observed by discussions on the mptcp-dev mailing list. In some cases the network operators would like to have a better control on how the subflows are created by Multipath TCP [I-D.boucadair-mptcp-max-subflow]. This might require the definition of policy rules to control the operation of the subflow manager. The two scenarios below illustrate some of these requirements.

```
           host1 ----------  switch1 ----- host2
             |                 |            |
             +--------------  switch2 --------+
```

Figure 3: Simple switched network topology

Consider the simple network topology shown in Figure 3. From an operational viewpoint, a network operator could want to create two subflows between the communicating hosts. From a bandwidth utilization viewpoint, the most natural paths are host1-switch1-host2 and host1-switch2-host2. However, a Multipath TCP implementation running on these two hosts may sometimes have difficulties to obtain this result.

To understand the difficulty, let us consider different allocation strategies for the IP addresses. A first strategy is to assign two subnets : subnetA (resp. subnetB) contains the IP addresses of host1’s interface to switch1 (resp. switch2) and host2’s interface to switch1 (resp. switch2). In this case, a Multipath TCP subflow manager should only create one subflow per subnet. To enforce the utilization of these paths, the network operator would have to specify a policy that prefers the subflows in the same subnet over subflows between addresses in different subnets. It should be noted that the policy should probably also specify how the subflow manager should react when an interface or subflow fails.

A second strategy is to use a single subnet for all IP addresses. In this case, it becomes more difficult to specify a policy that indicates which subflows should be established.

The second subflow manager that is currently supported by the Multipath TCP implementation in the Linux kernel is the ndiffport subflow manager. This manager was initially created to exploit the path diversity that exists between single-homed hosts due to the utilization of flow-based load balancing techniques [SIGCOMM11]. This subflow manager creates N subflows between the same pair of IP addresses. The N subflows are created by the client and differ only in the source port selected by the client. It was not designed to be used on multihomed hosts.

A more flexible subflow manager has been proposed, implemented and evaluated in [CONEXT15]. This subflow manager exposes various kernel events to a user space daemon that decides when subflows need to be created and terminated based on various policies.

3.5. Subflow destination port

The Multipath TCP protocol relies on the token contained in the MP_JOIN option to associate a subflow to an existing Multipath TCP session. This implies that there is no restriction on the source address, destination address and source or destination ports used for the new subflow. The ability to use different source and destination addresses is key to support multihomed servers and clients. The ability to use different destination port numbers is worth discussing because it has operational implications.

For illustration, consider a dual-homed client that creates a second subflow to reach a single-homed server as illustrated in Figure 4.

```
        client ------- r1 --- internet --- server
            |                   |
            +----------r2-------+
```

Figure 4: Multihomed-client connected to single-homed server

When the Multipath TCP implementation in the Linux kernel creates the second subflow it uses the same destination port as the initial subflow. This choice is motivated by the fact that the server might be protected by a firewall and only accept TCP connections (including subflows) on the official port number. Using the same destination port for all subflows is also useful for operators that rely on the port numbers to track application usage in their network.

There have been suggestions from Multipath TCP users to modify the implementation to allow the client to use different destination ports to reach the server. This suggestion seems mainly motivated by traffic shaping middleboxes that are used in some wireless networks. In networks where different shaping rates are associated to different destination port numbers, this could allow Multipath TCP to reach a higher performance. This behavior is valid according to the Multipath TCP specification [RFC6824]. An application could used an enhanced socket API [SOCKET] to behave in this way.

However, from an implementation point-of-view supporting different destination ports for the same Multipath TCP connection can cause some issues. A legacy implementation of a TCP stack creates a listening socket to react upon incoming SYN segments. The listening socket is handling the SYN segments that are sent on a specific port number. Demultiplexing incoming segments can thus be done solely by looking at the IP addresses and the port numbers. With Multipath TCP however, incoming SYN segments may have an MP_JOIN option with a different destination port. This means, that all incoming segments that did not match on an existing listening-socket or an already established socket must be parsed for an eventual MP_JOIN option. This imposes an additional cost on servers, previously not existent on legacy TCP implementations.

3.6. Closing subflows

```
                    client                       server
                       |                           |
   MPTCP: established  |                           | MPTCP: established
   Sub: established    |                           | Sub: established
                       |                           |
                       |         DATA_FIN          |
   MPTCP: close-wait   | <------------------------ | close()   (step 1)
   Sub: established    |         DATA_ACK          |
                       | ------------------------> | MPTCP: fin-wait-2
                       |                           | Sub: established
                       |                           |
                       |  DATA_FIN + subflow-FIN   |
   close()/shutdown()  | ------------------------> | MPTCP: time-wait
   (step 2)            |        DATA_ACK           | Sub: close-wait
   MPTCP: closed       | <------------------------ |
   Sub: fin-wait-2     |                           |
                       |                           |
                       |        subflow-FIN        |
   MPTCP: closed       | <------------------------ | subflow-close()
   Sub: time-wait      |        subflow-ACK        |
   (step 3)            | ------------------------> | MPTCP: time-wait
                       |                           | Sub: closed
                       |                           |
```

Figure 5: Multipath TCP may not be able to avoid time-wait state on the subflow (indicated as Sub in the drawing), even if enforced by the application on the client-side.

Figure 5 shows a very particular issue within Multipath TCP. Many high-performance applications try to avoid Time-Wait state by deferring the closure of the connection until the peer has sent a FIN. That way, the client on the left of Figure 5 does a passive closure of the connection, transitioning from Close-Wait to Last-ACK and finally freeing the resources after reception of the ACK of the FIN. An application running on top of a Multipath TCP enabled Linux kernel might also use this approach. The difference here is that the close() of the connection (Step 1 in Figure 5) only triggers the sending of a DATA_FIN. Nothing guarantees that the kernel is ready to combine the DATA_FIN with a subflow-FIN. The reception of the DATA_FIN will make the application trigger the closure of the connection (step 2), trying to avoid Time-Wait state with this late closure. This time, the kernel might decide to combine the DATA_FIN with a subflow-FIN. This decision will be fatal, as the subflow's state machine will not transition from Close-Wait to Last-Ack, but rather go through Fin-Wait-2 into Time-Wait state. The Time-Wait state will consume resources on the host for at least 2 MSL (Maximum Segment Lifetime). Thus, a smart application that tries to avoid Time-Wait state by doing late closure of the connection actually ends up with one of its subflows in Time-Wait state. A high-performance Multipath TCP kernel implementation should honor the desire of the application to do passive closure of the connection and successfully avoid Time-Wait state - even on the subflows.

The solution to this problem lies in an optimistic assumption that a host doing active-closure of a Multipath TCP connection by sending a DATA_FIN will soon also send a FIN on all its subflows. Thus, the passive closer of the connection can simply wait for the peer to send exactly this FIN - enforcing passive closure even on the subflows. Of course, to avoid consuming resources indefinitely, a timer must limit the time our implementation waits for the FIN.

3.7. Packet schedulers

In a Multipath TCP implementation, the packet scheduler is the algorithm that is executed when transmitting each packet to decide on which subflow it needs to be transmitted. The packet scheduler itself does not have any impact on the interoperability of Multipath TCP implementations. However, it may clearly impact the performance of Multipath TCP sessions. The Multipath TCP implementation in the Linux kernel supports a pluggable architecture for the packet scheduler [PaaschPhD]. As of this writing, two schedulers have been implemented: round-robin and lowest-rtt-first. The second scheduler relies on the round-trip-time (rtt) measured on each TCP subflow and sends first segments over the subflow having the lowest round-trip-time. They are compared in [CSWS14]. The experiments and measurements described in [CSWS14] show that the lowest-rtt-first scheduler appears to be the best compromise from a performance viewpoint. Another study of the packet schedulers is presented in [PAMS2014]. This study relies on simulations with the Multipath TCP implementation in the Linux kernel. They compare the lowest-rtt-first with the round-robin and a random scheduler. They show some situations where the lowest-rtt-first scheduler does not perform as well as the other schedulers, but there are many scenarios where the opposite is true. [PAMS2014] notes that "it is highly likely that the optimal scheduling strategy depends on the characteristics of the paths being used."

3.8. Segment size selection

When an application performs a write/send system call, the kernel allocates a packet buffer (sk_buff in Linux) to store the data the application wants to send. The kernel will store at most one MSS (Maximum Segment Size) of data per buffer. As the MSS can differ amongst subflows, an MPTCP implementation must select carefully the MSS used to generate application data. The Linux kernel implementation had various ways of selecting the MSS: minimum or maximum amongst the different subflows. However, these heuristics of MSS selection can cause significant performance issues in some environment. Consider the following example. An MPTCP connection has two established subflows that respectively use a MSS of 1420 and 1428 bytes. If MPTCP selects the maximum, then the application will generate segments of 1428 bytes of data. An MPTCP implementation will have to split the segment in two (1420-byte and 8-byte) segments when pushing on the subflow with the smallest MSS. The latter segment will introduce a large overhead as for a single data segment 2 slots will be used in the congestion window (in packets) therefore reducing by roughly twice the potential throughput (in bytes/s) of this subflow. Taking the smallest MSS does not solve the issue as there might be a case where the subflow with the smallest MSS only sends a few packets therefore reducing the potential throughput of the other subflows.

The Linux implementation recently took another approach [DetalMSS]. Instead of selecting the minimum and maximum values, it now dynamically adapts the MSS based on the contribution of all the subflows to the connection’s throughput. For this it computes, for each subflow, the potential throughput achieved by selecting each MSS value and by taking into account the lost space in the congestion window. It then selects the MSS that allows to achieve the highest potential throughput.

Given the prevalence of middleboxes that clamp the MSS, Multipath TCP implementations must be able to efficiently support subflows with different MSS values. The strategy described above is a possible solution to this problem.

3.9. Interactions with the Domain Name System

Multihomed clients such as smartphones can send DNS queries over any of their interfaces. When a single-homed client performs a DNS query, it receives from its local resolver the best answer for its request. If the client is multihomed, the answer in response to the DNS query may vary with the interface over which it has been sent.

```
                      cdn1
                       |
           client -- cellular -- internet -- cdn3
              |                   |
              +----- wifi --------+
                       |
                     cdn2
```

Figure 6: Simple network topology

If the client sends a DNS query over the WiFi interface, the answer will point to the cdn2 server while the same request sent over the cellular interface will point to the cdn1 server. This might cause problems for CDN providers that locate their servers inside ISP networks and have contracts that specify that the CDN server will only be accessed from within this particular ISP. Assume now that both the client and the CDN servers support Multipath TCP. In this case, a Multipath TCP session from cdn1 or cdn2 would potentially use both the cellular network and the WiFi network. Serving the client from cdn2 over the cellular interface could violate the contract between the CDN provider and the network operators. A similar problem occurs with regular TCP if the client caches DNS replies. For example the client obtains a DNS answer over the cellular interface and then stops this interface and starts to use its WiFi interface. If the client retrieves data from cdn1 over its WiFi interface, this may also violate the contract between the CDN and the network operators.

A possible solution to prevent this problem would be to modify the DNS resolution on the client. The client subnet EDNS extension defined in [RFC7871] could be used for this purpose. When the client sends a DNS query from its WiFi interface, it should also send the client subnet corresponding to the cellular interface in this request. This would indicate to the resolver that the answer should be valid for both the WiFi and the cellular interfaces (e.g., the cdn3 server).

3.10. Captive portals

Multipath TCP enables a host to use different interfaces to reach a server. In theory, this should ensure connectivity when at least one of the interfaces is active. In practice however, there are some particular scenarios with captive portals that may cause operational problems. The reference environment is shown in Figure 7.

```
           client -----  network1
                |
                +------- internet ------------- server
```

Figure 7: Issue with captive portal

The client is attached to two networks : network1 that provides limited connectivity and the entire Internet through the second network interface. In practice, this scenario corresponds to an open WiFi network with a captive portal for network1 and a cellular service for the second interface. On many smartphones, the WiFi interface is preferred over the cellular interface. If the smartphone learns a default route via both interfaces, it will typically prefer to use the WiFi interface to send its DNS request and create the first subflow. This is not optimal with Multipath TCP. A better approach would probably be to try a few attempts on the WiFi interface and then, upon failure of these attempts, try to use the second interface for the initial subflow as well.

3.11. Stateless webservers

MPTCP has been designed to interoperate with webservers that benefit from SYN-cookies to protect against SYN-flooding attacks [RFC4987]. MPTCP achieves this by echoing the keys negotiated during the MP_CAPABLE handshake in the third ACK of the 3-way handshake. Reception of this third ACK then allows the server to reconstruct the state specific to MPTCP.

However, one caveat to this mechanism is the non-reliable nature of the third ACK. Indeed, when the third ACK gets lost, the server will not be able to reconstruct the MPTCP-state. MPTCP will fallback to regular TCP in this case. This is in contrast to regular TCP. When the client starts sending data, the first data segment also includes the SYN-cookie, which allows the server to reconstruct the TCP-state. Further, this data segment will be retransmitted by the client in case it gets lost and thus is resilient against loss. MPTCP does not include the keys in this data segment and thus the server cannot reconstruct the MPTCP state.

This issue might be considered as a minor one for MPTCP. Losing the third ACK should only happen when packet loss is high. However, when packet-loss is high MPTCP provides a lot of benefits as it can move traffic away from the lossy link. It is undesirable that MPTCP has a higher chance to fall back to regular TCP in those lossy environments.

[I-D.paasch-mptcp-syncookies] discusses this issue and suggests a modified handshake mechanism that ensures reliable delivery of the MP_CAPABLE, following the 3-way handshake. This modification will make MPTCP reliable, even in lossy environments when servers need to use SYN-cookies to protect against SYN-flooding attacks.

3.12. Loadbalanced server farms

Large-scale server farms typically deploy thousands of servers behind a single virtual IP (VIP). Steering traffic to these servers is done through layer-4 load balancers that ensure that a TCP-flow will always be routed to the same server [Presto08].

As Multipath TCP uses multiple different TCP subflows to steer the traffic across the different paths, load balancers need to ensure that all these subflows are routed to the same server. This implies that the load balancers need to track the MPTCP-related state, allowing them to parse the token in the MP_JOIN and assign those subflows to the appropriate server. However, server farms typically deploy several load balancers for reliability and capacity reasons. As a TCP subflow might get routed to any of these load balancers, they would need to synchronize the MPTCP-related state - a solution that is not feasible at large scale.

The token (carried in the MP_JOIN) contains the information indicating which MPTCP-session the subflow belongs to. As the token is a hash of the key, servers are not able to generate the token in such a way that the token can provide the necessary information to the load balancers, which would allow them to route TCP subflows to the appropriate server. [I-D.paasch-mptcp-loadbalancer] discusses this issue in detail and suggests two alternative MP_CAPABLE handshakes to overcome these.

4. IANA Considerations

There are no IANA considerations in this informational document.

5. Security Considerations

This informational document discusses use-cases and operational experience with Multipath TCP. An extensive analysis of the remaining security issues in the Multipath TCP specification has been published in [RFC7430], together with suggestions for possible solutions.

From a security viewpoint, it is important to note that Multipath TCP, like other multipath solutions such as SCTP, has the ability to send packets belonging to a single connection over different paths. This design feature of Multipath TCP implies that middleboxes that have been deployed on-path assuming that they would observe all the packets exchanged for a given connection in both directions may not function correctly anymore. A typical example are firewalls, IDS or DPIs deployed in enterprise networks. Those devices expect to observe all the packets from all TCP connections. With Multipath TCP, those middleboxes may not observe anymore all packets since some of them may follow a different path. The two examples below illustrate typical deployments of such middleboxes. The first example, Figure 8, shows a Multipath TCP enabled smartphone attached to both an enterprise and a cellular network. If a Multipath TCP connection is established by the smartphone towards a server, some of the packets sent by the smartphone or the server may be transmitted over the cellular network and thus be invisible for the enterprise middlebox.

```
smartphone +----- entreprise net --- MBox----+------ server
           |                                 |
           +----- cellular net  -------------+
```

Figure 8: Enterprise Middlebox may not observe all packets from multihomed host

The second example, Figure 9, shows a possible issue when multiple middleboxes are deployed inside a network. For simplicity, we assume that network1 is the default IPv4 path while network2 is the default IPv6 path. A similar issue could occur with per-flow load balancing such as ECMP [RFC2992]. With regular TCP, all packets from each connection would either pass through Mbox1 or Mbox2. With Multipath TCP, the client can easily establish a subflow over network1 and another over network2 and each middlebox would only observe a part of the traffic of the end-to-end Multipath TCP connection.

```
     client ----R-- network1  --- MBox1 -----R------------- server
                |                            |
                +-- network2  --- MBox2 -----+
```

Figure 9: Interactions between load balancing and security Middleboxes

In these two cases, it is possible for an attacker to evade some security measures operating on the TCP byte stream and implemented on the middleboxes by controlling the bytes that are actually sent over each subflow and there are tools that ease those kinds of evasion [PZ15] [PT14]. This is not a security issue for Multipath TCP itself since Multipath TCP behaves correctly. However, this demonstrates the difficulty of enforcing security policies by relying only on on-path middleboxes instead of enforcing them directly on the endpoints.

6. Acknowledgements

This work was partially supported by the FP7-Trilogy2 project. We would like to thank all the implementers and users of the Multipath TCP implementation in the Linux kernel. This document has benefited from the comments of John Ronan, Yoshifumi Nishida, Phil Eardley, Jaehyun Hwang, Mirja Kuehlewind, Benoit Claise, Jari Arkko, Qin Wu, Spencer Dawkins and Ben Campbell.

7. References

7.1. Normative References

[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>.

[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>.

7.2. Informative References

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

[BALIA] Peng, Q., Walid, A., Hwang, J., and S. Low, "Multipath TCP Analysis, Design, and Implementation", IEEE/ACM Trans. on Networking, Vol. 24, No. 1 , 2016.

[CACM14] Paasch, C. and O. Bonaventure, "Multipath TCP", Communications of the ACM, 57(4):51-57 , April 2014, <http://inl.info.ucl.ac.be/publications/multipath-tcp>.

[COMCOM2016] Tran, V., De Coninck, Q., Hesmans, B., Sadre, R., and O. Bonaventure, "Observing real Multipath TCP traffic", Computer Communications , April 2016, <http://inl.info.ucl.ac.be/publications/observing-real-multipath-tcp-traffic>.

[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>.

[CONEXT12] Khalili, R., Gast, N., Popovic, M., Upadhyay, U., and J. Leboudec, "MPTCP is not pareto-optimal performance issues and a possible solution", Proceedings of the 8th international conference on Emerging networking experiments and technologies (CoNEXT12) , 2012.

[CONEXT13]
Paasch, C., Khalili, R., and O. Bonaventure, "On the Benefits of Applying Experimental Design to Improve Multipath TCP", Conference on emerging Networking EXperiments and Technologies (CoNEXT) , December 2013, <http://inl.info.ucl.ac.be/publications/benefits-applying-experimental-design-improve-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>.

[CSWS14] Paasch, C., Ferlin, S., Alay, O., and O. Bonaventure, "Experimental Evaluation of Multipath TCP Schedulers", SIGCOMM CSWS2014 workshop , August 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>.

[DetalMSS]
Detal, G., "Adaptive MSS value", Post on the mptcp-dev mailing list , September 2014, <https://listes-2.sipr.ucl.ac.be/sympa/arc/mptcp-dev/2014-09/msg00130.html>.

[FreeBSD-MPTCP]
Williams, N., "Multipath TCP For FreeBSD Kernel Patch v0.5", n.d., <http://caia.swin.edu.au/urp/newtcp/mptcp>.

[HAMPEL] Hampel, G., Rana, A., and T. Klein, "Seamless TCP mobility using lightweight MPTCP proxy", Proceedings of the 11th ACM international symposium on Mobility management and wireless access (MobiWac ’13). , 2013.

[HotMiddlebox13]
Hesmans, B., Duchene, F., Paasch, C., Detal, G., and O. Bonaventure, "Are TCP Extensions Middlebox-proof?", CoNEXT workshop HotMiddlebox , December 2013, <http://

inl.info.ucl.ac.be/publications/are-tcp-extensions-middlebox-proof>.

[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>.

[HotNets] Raiciu, C., Pluntke, C., Barre, S., Greenhalgh, A., Wischik, D., and M. Handley, "Data center networking with multipath TCP", Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets-IX) , 2010, <http://doi.acm.org/10.1145/1868447.1868457>.

[I-D.boucadair-mptcp-max-subflow]
Boucadair, M. and C. Jacquenet, "Negotiating the Maximum Number of Multipath TCP (MPTCP) Subflows", draft-boucadair-mptcp-max-subflow-02 (work in progress), May 2016.

[I-D.lhwxz-gre-notifications-hybrid-access]
Leymann, N., Heidemann, C., Wasserman, M., Xue, L., and M. Zhang, "GRE Notifications for Hybrid Access", draft-lhwxz-gre-notifications-hybrid-access-01 (work in progress), January 2015.

[I-D.lhwxz-hybrid-access-network-architecture]
Leymann, N., Heidemann, C., Wasserman, M., Xue, L., and M. Zhang, "Hybrid Access Network Architecture", draft-lhwxz-hybrid-access-network-architecture-02 (work in progress), January 2015.

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

[I-D.paasch-mptcp-syncookies]
Paasch, C., Biswas, A., and D. Haas, "Making Multipath TCP robust for stateless webservers", draft-paasch-mptcp-syncookies-02 (work in progress), October 2015.

[ICNP12] Cao, Y., Xu, M., and X. Fu, "Delay-based congestion control for multipath TCP", 20th IEEE International Conference on Network Protocols (ICNP) , 2012.

[IETFJ] Bonaventure, O. and S. Seo, "Multipath TCP Deployments,", IETF Journal, Vol. 12, Issue 2 , November 2016.

[IMC11] Honda, M., Nishida, Y., Raiciu, C., Greenhalgh, A., Handley, M., and H. Tokuda, "Is it still possible to extend TCP?", Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference (IMC ’11) , 2011, <http://doi.acm.org/10.1145/2068816.2068834>.

[IMC13a] Detal, G., Hesmans, B., Bonaventure, O., Vanaubel, Y., and B. Donnet, "Revealing Middlebox Interference with Tracebox", Proceedings of the 2013 ACM SIGCOMM conference on Internet measurement conference , 2013, <http://inl.info.ucl.ac.be/publications/revealing-middlebox-interference-tracebox>.

[IMC13b] Chen, Y., Lim, Y., Gibbens, R., Nahum, E., Khalili, R., and D. Towsley, "A measurement-based study of MultiPath TCP performance over wireless network", Proceedings of the 2013 conference on Internet measurement conference (IMC ’13) , n.d., <http://doi.acm.org/10.1145/2504730.2504751>.

[IMC13c] Pelsser, C., Cittadini, L., Vissicchio, S., and R. Bush, "From Paris to Tokyo on the suitability of ping to measure latency", Proceedings of the 2013 conference on Internet measurement conference (IMC ’13) , 2013, <http://doi.acm.org/10.1145/2504730.2504765>.

[INFOCOM14]
Lim, Y., Chen, Y., Nahum, E., Towsley, D., and K. Lee, "Cross-Layer Path Management in Multi-path Transport Protocol for Mobile Devices", IEEE INFOCOM’14 , 2014.

[IOS7] Apple, ., "Multipath TCP Support in iOS 7", January 2014, <http://support.apple.com/kb/HT5977>.

[KT] Seo, S., "KT’s GiGA LTE", July 2015, <https://www.ietf.org/proceedings/93/slides/slides-93-mptcp-3.pdf>.

[MBTest] Hesmans, B., "MBTest", 2013, <https://bitbucket.org/bhesmans/mbtest>.

[MPTCPBIB]
Bonaventure, O., "Multipath TCP - An annotated bibliography", Technical report , April 2015, <https://github.com/obonaventure/mptcp-bib>.

[Mobicom15]

De Coninck, Q., Baerts, M., Hesmans, B., and O. Bonaventure, "Poster - Evaluating Android Applications with Multipath TCP", Mobicom 2015 (Poster) , September 2015.

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

[NSDI11] Wischik, D., Raiciu, C., Greenhalgh, A., and M. Handley, "Design, implementation and evaluation of congestion control for Multipath TCP", In Proceedings of the 8th USENIX conference on Networked systems design and implementation (NSDI11) , 2011.

[NSDI12] 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 of Networked Systems Design and Implementation (NSDI12) , April 2012, <http://inl.info.ucl.ac.be/publications/how-hard-can-it-be-designing-and-implementing-deployable-multipath-tcp>.

[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>.

[PAMS2014]
Arzani, B., Gurney, A., Cheng, S., Guerin, R., and B. Loo, "Impact of Path Selection and Scheduling Policies on MPTCP Performance", PAMS2014 , 2014.

[PT14] Pearce, C. and P. Thomas, "Multipath TCP Breaking Today’s Networks with Tomorrow’s Protocols", Proc. Blackhat Briefings , 2014, <http://www.blackhat.com/docs/us-14/materials/us-14-Pearce-Multipath-TCP-Breaking-Todays-Networks-With-Tomorrows-Protocols-WP.pdf>.

[PZ15] Pearce, C. and S. Zeadally, "Ancillary Impacts of Multipath TCP on Current and Future Network Security", IEEE Internet Computing, vol. 19, no. 5, pp. 58-65 , 2015.

[PaaschPhD]
Paasch, C., "Improving Multipath TCP", Ph.D. Thesis , November 2014, <http://inl.info.ucl.ac.be/publications/improving-multipath-tcp>.

[Presto08]
Greenberg, A., Lahiri, P., Maltz, D., Parveen, P., and S. Sengupta, "Towards a Next Generation Data Center Architecture - Scalability and Commoditization", ACM PRESTO 2008 , August 2008, <http://dl.acm.org/citation.cfm?id=1397732>.

[RFC1812] Baker, F., Ed., "Requirements for IP Version 4 Routers", RFC 1812, DOI 10.17487/RFC1812, June 1995, <http://www.rfc-editor.org/info/rfc1812>.

[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>.

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

[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>.

[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>.

[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>.

[RFC7871] Contavalli, C., van der Gaast, W., Lawrence, D., and W. Kumari, "Client Subnet in DNS Queries", RFC 7871, DOI 10.17487/RFC7871, May 2016, <http://www.rfc-editor.org/info/rfc7871>.

[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 , n.d., <http://doi.acm.org/10.1145/2018436.2018467>.

[SOCKET] Hesmans, B. and O. Bonaventure, "An Enhanced Socket API for Multipath TCP", Proceedings of the 2016 Applied Networking Research Workshop , July 2016, <http://doi.acm.org/10.1145/2959424.2959433>.

[StrangeMbox] Bonaventure, O., "Multipath TCP through a strange middlebox", Blog post , January 2015, <http://blog.multipath-tcp.org/blog/html/2015/01/30/multipath_tcp_through_a_strange_middlebox.html>.

[TMA2015] Hesmans, B., Tran Viet, H., Sadre, R., and O. Bonaventure, "A First Look at Real Multipath TCP Traffic", Traffic Monitoring and Analysis , 2015, <http://inl.info.ucl.ac.be/publications/first-look-real-multipath-tcp-traffic>.

[TR-348] Broadband Forum, ., "TR 348 - Hybrid Access Broadband Network Architecture", July 2016, <https://www.broadband-forum.org/technical/download/TR-348.pdf>.

[ietf88] Stewart, L., "IETF’88 Meeting minutes of the MPTCP working group", n.d., <http://tools.ietf.org/wg/mptcp/minutes?item=minutes-88-mptcp.html>.

[tracebox] Detal, G. and O. Tilmans, "tracebox", 2013, <http://www.tracebox.org>.

Appendix A. Changelog

This section should be removed before final publication

- initial version : September 16th, 2014 : Added section Section 3.8 that discusses some performance problems that appeared with the Linux implementation when using subflows having different MSS values
- update with a description of the middlebox that replaces an unknown TCP option with EOL [StrangeMbox]
- version ietf-02 : July 2015, answer to last call comments
  - Reorganised text to better separate use cases and operational experience
  - New use case on Multipath TCP proxies in Section 2.3
  - Added some text on middleboxes in Section 3.1
  - Removed the discussion on SDN
  - Restructured text and improved writing in some parts
- version ietf-03 : September 2015, answer to comments from Phil Eardley
  - Improved introduction
  - Added details about using SOCKS and Korea Telecom’s use-case in Section 2.3.
  - Added issue around clients caching DNS-results in Section 3.9
  - Explained issue of MPTCP with stateless webservers Section 3.11
  - Added description of MPTCP’s use behind layer-4 load balancers Section 3.12
  - Restructured text and improved writing in some parts
- version ietf-04 : April 2016, answer to last comments
  - Updated text on measurements with smartphones
  - Updated conclusion

- version ietf-06 : August 2016, answer to AD’s review

  - removed some expired drafts

  - removed conclusion

- version ietf-07 : September 2016, answer to IESG comments

  - small nits

  - added more information in the security considerations as suggested by Stephen Farrel

Authors’ Addresses

Olivier Bonaventure
UCLouvain

Email: Olivier.Bonaventure@uclouvain.be

Christoph Paasch
Apple, Inc.

Email: cpaasch@apple.com

Gregory Detal
Tessares

Email: Gregory.Detal@tessares.net

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

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

June 8, 2019

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

Abstract

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

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

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

Status of This Memo

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

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

Copyright Notice

Table of Contents

1. Introduction

Multipath TCP (MPTCP) is a set of extensions to regular TCP [RFC0793] to provide a Multipath TCP [RFC6182] service, which enables a transport connection to operate across multiple paths simultaneously. This document presents the protocol changes required to add multipath capability to TCP; specifically, those for signaling and setting up multiple paths ("subflows"), managing these subflows, reassembly of data, and termination of sessions. This is not the only information required to create a Multipath TCP implementation, however. This document is complemented by three others:

- o Architecture [RFC6182], which explains the motivations behind Multipath TCP, contains a discussion of high-level design decisions on which this design is based, and an explanation of a functional separation through which an extensible MPTCP implementation can be developed.

- o Congestion control [RFC6356] presents a safe congestion control algorithm for coupling the behavior of the multiple paths in order to "do no harm" to other network users.

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

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

1.1. Design Assumptions

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

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

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

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

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

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

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

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

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

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

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

1.2. Multipath TCP in the Networking Stack

MPTCP operates at the transport layer and aims to be transparent to both higher and lower layers. It is a set of additional features on top of standard TCP; Figure 1 illustrates this layering. MPTCP is designed to be usable by legacy applications with no changes; detailed discussion of its interactions with applications is given in [RFC6897].

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

Figure 1: Comparison of Standard TCP and MPTCP Protocol Stacks

1.3. Terminology

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

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

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

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

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

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

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

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

1.4. MPTCP Concept

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

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

- An MPTCP connection begins similarly to a regular TCP connection. This is illustrated in Figure 2 where an MPTCP connection is established between addresses A1 and B1 on Hosts A and B, respectively.

- If extra paths are available, additional TCP sessions (termed MPTCP "subflows") are created on these paths, and are combined with the existing session, which continues to appear as a single connection to the applications at both ends. The creation of the additional TCP session is illustrated between Address A2 on Host A and Address B1 on Host B.

- MPTCP identifies multiple paths by the presence of multiple addresses at hosts. Combinations of these multiple addresses equate to the additional paths. In the example, other potential paths that could be set up are A1<->B2 and A2<->B2. Although this additional session is shown as being initiated from A2, it could equally have been initiated from B1 or B2.

- The discovery and setup of additional subflows will be achieved through a path management method; this document describes a mechanism by which a host can initiate new subflows by using its own additional addresses, or by signaling its available addresses to the other host.

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

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

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

Figure 2: Example MPTCP Usage Scenario

1.5. Requirements Language

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

2. Operation Overview

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

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

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

2.1. Initiating an MPTCP Connection

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

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

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

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


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

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


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

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

2.2. Associating a New Subflow with an Existing MPTCP Connection

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

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

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

                      <-                ACK
```

2.3. Informing the Other Host about Another Potential Address

The set of IP addresses associated to a multihomed host may change during the lifetime of an MPTCP connection. MPTCP supports the addition and removal of addresses on a host both implicitly and explicitly. If Host A has established a subflow starting at address/ port pair IP#-A1 and wants to open a second subflow starting at address/port pair IP#-A2, it simply initiates the establishment of the subflow as explained above. The remote host will then be implicitly informed about the new address.

In some circumstances, a host may want to advertise to the remote host the availability of an address without establishing a new subflow, for example, when a NAT prevents setup in one direction. In the example below, Host A informs Host B about its alternative IP address/port pair (IP#-A2). Host B may later send an MP_JOIN to this new address. The ADD_ADDR option contains a HMAC to authenticate the address as having been sent from the originator of the connection. The receiver of this option echoes it back to the client to indicate successful receipt. Further details are in Section 3.4.1.

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

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

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

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

2.4. Data Transfer Using MPTCP

To ensure reliable, in-order delivery of data over subflows that may appear and disappear at any time, MPTCP uses a 64-bit data sequence number (DSN) to number all data sent over the MPTCP connection. Each subflow has its own 32-bit sequence number space, utilising the regular TCP sequence number header, and an MPTCP option maps the subflow sequence space to the data sequence space. In this way, data can be retransmitted on different subflows (mapped to the same DSN) in the event of failure.

The Data Sequence Signal (DSS) carries the Data Sequence Mapping. The Data Sequence Mapping consists of the subflow sequence number, data sequence number, and length for which this mapping is valid. This option can also carry a connection-level acknowledgment (the "Data ACK") for the received DSN.

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

Further details are in Section 3.3.

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

2.5. Requesting a Change in a Path’s Priority

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

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

2.6. Closing an MPTCP Connection

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

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

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

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

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

   [RST on all other 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.

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

Figure 3: MPTCP Option Format

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

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

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

3.1. Connection Initiation

Connection initiation begins with a SYN, SYN/ACK, ACK exchange on a single path. Each packet contains the Multipath Capable (MP_CAPABLE) MPTCP option (Figure 4). This option declares its sender is capable of performing Multipath TCP and wishes to do so on this particular connection.

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

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

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

Figure 4: Multipath Capable (MP_CAPABLE) Option

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

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

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

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

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

The contents of the option is determined by the SYN and ACK flags of the packet, along with the option’s length field. For the diagram shown in Figure 4, "sender" and "receiver" refer to the sender or receiver of the TCP packet (which can be either host).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.2. Starting a New Subflow

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 8: Example Use of MPTCP Authentication

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

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

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

If the HMACs are verified as correct, then both hosts have verified each other as being the same peers as existed at the start of the connection, and they have agreed of which connection this subflow will become a part.

If the SYN/ACK as received at Host A does not have an MP_JOIN option, Host A MUST close the subflow with a TCP RST.

This covers all cases of the loss of an MP_JOIN. In more detail, if MP_JOIN is stripped from the SYN on the path from A to B, and Host B does not have a listener on the relevant port, it will respond with a RST in the normal way. If in response to a SYN with an MP_JOIN option, a SYN/ACK is received without the MP_JOIN option (either since it was stripped on the return path, or it was stripped on the outgoing path but Host B responded as if it were a new regular TCP session), then the subflow is unusable and Host A MUST close it with a RST.

Note that additional subflows can be created between any pair of ports (but see Section 3.9 for heuristics); no explicit application-level accept calls or bind calls are required to open additional subflows. To associate a new subflow with an existing connection, the token supplied in the subflow's SYN exchange is used for demultiplexing. This then binds the 5-tuple of the TCP subflow to the local token of the connection. A consequence is that it is possible to allow any port pairs to be used for a connection.

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

3.3. General MPTCP Operation

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

The data sequence mapping and the Data ACK are signaled in the Data Sequence Signal (DSS) option (Figure 9). Either or both can be signaled in one DSS, depending on the flags set. The data sequence mapping defines how the sequence space on the subflow maps to the connection level, and the Data ACK acknowledges receipt of data at the connection level. These functions are described in more detail in the following two subsections.

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

Figure 9: Data Sequence Signal (DSS) Option

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

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.

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

Figure 10: Pseudo-Header for DSS Checksum

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

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

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

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

Data sequence numbers are always 64-bit quantities, and MUST be maintained as such in implementations. If a connection is progressing at a slow rate, so protection against wrapped sequence numbers is not required, then an implementation MAY include just the lower 32 bits of the data sequence number in the data sequence mapping and/or Data ACK as an optimization, and an implementation can make this choice independently for each packet. An implementation MUST be able to receive and process both 64-bit or 32-bit sequence number values, but it is not required that an implementation is able to send both.

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

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

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

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

3.3.2. Data Acknowledgments

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

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

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

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

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

3.3.3. Closing a Connection

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

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

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

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

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

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

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

A connection is considered closed once both hosts' DATA_FINs have been acknowledged by DATA_ACKs.

As specified above, a standard TCP FIN on an individual subflow only shuts down the subflow on which it was sent. If all subflows have been closed with a FIN exchange, but no DATA_FIN has been received and acknowledged, the MPTCP connection is treated as closed only after a timeout. This implies that an implementation will have TIME_WAIT states at both the subflow and connection levels (see Appendix D). This permits "break-before-make" scenarios where connectivity is lost on all subflows before a new one can be re-established.

3.3.4. Receiver Considerations

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

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

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

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

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

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

3.3.5. Sender Considerations

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

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

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

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

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

3.3.6. Reliability and Retransmissions

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

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

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

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

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

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

3.3.7. Congestion Control Considerations

Different subflows in an MPTCP connection have different congestion windows. To achieve fairness at bottlenecks and resource pooling, it is necessary to couple the congestion windows in use on each subflow, in order to push most traffic to uncongested links. One algorithm for achieving this is presented in [RFC6356]; the algorithm does not achieve perfect resource pooling but is "safe" in that it is readily deployable in the current Internet. By this, we mean that it does not take up more capacity on any one path than if it was a single path flow using only that route, so this ensures fair coexistence with single-path TCP at shared bottlenecks.

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

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

3.3.8. Subflow Policy

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

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

For instance, a possibility is an ’all-or-nothing’ approach, i.e., have a second path ready for use in the event of failure of the first path, but alternatives could include entirely saturating one path before using an additional path (the ’overflow’ case). Such choices would be most likely based on the monetary cost of links, but may also be based on properties such as the delay or jitter of links, where stability (of delay or bandwidth) is more important than throughput. Application requirements such as these are discussed in detail in [RFC6897].

The ability to make effective choices at the sender requires full knowledge of the path "cost", which is unlikely to be the case. It would be desirable for a receiver to be able to signal their own preferences for paths, since they will often be the multihomed party, and may have to pay for metered incoming bandwidth.

To enable this, the MP_JOIN option (see Section 3.2) contains the 'B' bit, which allows a host to indicate to its peer that this path should be treated as a backup path to use only in the event of failure of other working subflows (i.e., a subflow where the receiver has indicated B=1 SHOULD NOT be used to send data unless there are no usable subflows where B=0).

In the event that the available set of paths changes, a host may wish to signal a change in priority of subflows to the peer (e.g., a subflow that was previously set as backup should now take priority over all remaining subflows). Therefore, the MP_PRIO option, shown in Figure 11, can be used to change the 'B' flag of the subflow on which it is sent.

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

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

Figure 11: Change Subflow Priority (MP_PRIO) Option

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

3.4. Address Knowledge Exchange (Path Management)

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

This design makes use of two methods of sharing such information, and both can be used on a connection. The first is the direct setup of new subflows, already described in Section 3.2, where the initiator has an additional address. The second method, described in the following subsections, signals addresses explicitly to the other host to allow it to initiate new subflows. The two mechanisms are complementary: the first is implicit and simple, while the explicit is more complex but is more robust. Together, the mechanisms allow addresses to change in flight (and thus support operation through NATs, since the source address need not be known), and also allow the signaling of previously unknown addresses, and of addresses belonging to other address families (e.g., both IPv4 and IPv6).

Here is an example of typical operation of the protocol:

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

- Simultaneously (or after a timeout), an ADD_ADDR option (Section 3.4.1) is sent on an existing subflow, informing the receiver of the sender’s alternative address(es). The recipient can use this information to open a new subflow to the sender’s additional address. In our example, A will send ADD_ADDR option informing B of address/port A2. The mix of using the SYN-based option and the ADD_ADDR option, including timeouts, is implementation specific and can be tailored to agree with local policy.

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

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

3.4.1. Address Advertisement

The Add Address (ADD_ADDR) MPTCP option announces additional addresses (and optionally, ports) on which a host can be reached (Figure 12). This option can be used at any time during a connection, depending on when the sender wishes to enable multiple paths and/or when paths become available. As with all MPTCP signals, the receiver MUST undertake standard TCP validity checks, e.g. [RFC5961], before acting upon it.

Every address has an Address ID that can be used for uniquely identifying the address within a connection for address removal. The Address ID is also used to identify MP_JOIN options (see Section 3.2) relating to the same address, even when address translators are in use. The Address ID MUST uniquely identify the address for the sender of the option (within the scope of the connection), but the mechanism for allocating such IDs is implementation specific.

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

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

The 2 octets that specify the TCP port number to use are optional and their presence can be inferred from the length of the option. Although it is expected that the majority of use cases will use the same port pairs as used for the initial subflow (e.g., port 80 remains port 80 on all subflows, as does the ephemeral port at the client), there may be cases (such as port-based load balancing) where the explicit specification of a different port is required. If no port is specified, MPTCP SHOULD attempt to connect to the specified address on the same port as is already in use by the subflow on which the ADD_ADDR signal was sent; this is discussed in more detail in Section 3.9.

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

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

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

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

Figure 12: Add Address (ADD_ADDR) Option

Due to the proliferation of NATs, it is reasonably likely that one host may attempt to advertise private addresses [RFC1918]. It is not desirable to prohibit this, since there may be cases where both hosts have additional interfaces on the same private network, and a host MAY advertise such addresses. The MP_JOIN handshake to create a new subflow (Section 3.2) provides mechanisms to minimize security risks. The MP_JOIN message contains a 32-bit token that uniquely identifies the connection to the receiving host. If the token is unknown, the host will return with a RST. In the unlikely event that the token is valid at the receiving host, subflow setup will continue, but the HMAC exchange must occur for authentication. This will fail, and will provide sufficient protection against two unconnected hosts accidentally setting up a new subflow upon the signal of a private address. Further security considerations around the issue of ADD_ADDR messages that accidentally misdirect, or maliciously direct, new MP_JOIN attempts are discussed in Section 5.

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

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

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

During normal MPTCP operation, it is unlikely that there will be sufficient TCP option space for ADD_ADDR to be included along with those for data sequence numbering (Section 3.3.1). Therefore, it is expected that an MPTCP implementation will send the ADD_ADDR option on separate ACKs. As discussed earlier, however, an MPTCP implementation MUST NOT treat duplicate ACKs with any MPTCP option, with the exception of the DSS option, as indications of congestion [RFC5681], and an MPTCP implementation SHOULD NOT send more than two duplicate ACKs in a row for signaling purposes.

3.4.2. Remove Address

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

This is achieved through the Remove Address (REMOVE_ADDR) option (Figure 13), which will remove a previously added address (or list of addresses) from a connection and terminate any subflows currently using that address.

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

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

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

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

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

Figure 13: Remove Address (REMOVE_ADDR) Option

3.5. Fast Close

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

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

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

Figure 14: Fast Close (MP_FASTCLOSE) Option

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

- Option A (ACK) : Host A sends an ACK containing the MP_FASTCLOSE option on one subflow, containing the key of Host B as declared in the initial connection handshake. On all the other subflows, Host A sends a regular TCP RST to close these subflows, and tears them down. Host A now enters FASTCLOSE_WAIT state.
- Option R (RST) : Host A sends a RST containing the MP_FASTCLOSE option on all subflows, containing the key of Host B as declared in the initial connection handshake. Host A can tear the subflows and the connection down immediately.

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

- Upon receipt of an ACK with MP_FASTCLOSE by Host B, containing the valid key, Host B answers on the same subflow with a TCP RST and tears down all subflows also through sending TCP RST signals. Host B can now close the whole MPTCP connection (it transitions directly to CLOSED state).
- As soon as Host A has received the TCP RST on the remaining subflow, it can close this subflow and tear down the whole connection (transition from FASTCLOSE_WAIT to CLOSED states). If Host A receives an MP_FASTCLOSE instead of a TCP RST, both hosts attempted fast closure simultaneously. Host A should reply with a TCP RST and tear down the connection.
- If Host A does not receive a TCP RST in reply to its MP_FASTCLOSE after one retransmission timeout (RTO) (the RTO of the subflow where the MP_FASTCLOSE has been sent), it SHOULD retransmit the MP_FASTCLOSE. The number of retransmissions SHOULD be limited to avoid this connection from being retained for a long time, but

this limit is implementation specific. A RECOMMENDED number is 3. If no TCP RST is received in response, Host A SHOULD send a TCP RST with the MP_FASTCLOSE option itself when it releases state in order to clear any remaining state at middleboxes.

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

3.6. Subflow Reset

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

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

Figure 15: TCP RST Reason (MP_TCPRST) Option

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

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

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

- 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.

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

Figure 16: Fallback (MP_FAIL) Option

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

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

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

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

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

It should be emphasized that MPTCP is not attempting to prevent the use of middleboxes that want to adjust the payload. An MPTCP-aware middlebox could provide such functionality by also rewriting checksums.

3.8. Error Handling

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

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

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

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

- Remove request for unknown address ID: silently ignore

3.9. Heuristics

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

3.9.1. Port Usage

Under typical operation, an MPTCP implementation SHOULD use the same ports as already in use. In other words, the destination port of a SYN containing an MP_JOIN option SHOULD be the same as the remote port of the first subflow in the connection. The local port for such SYNs SHOULD also be the same as for the first subflow (and as such, an implementation SHOULD reserve ephemeral ports across all local IP addresses), although there may be cases where this is infeasible. This strategy is intended to maximize the probability of the SYN being permitted by a firewall or NAT at the recipient and to avoid confusing any network monitoring software.

There may also be cases, however, where a host wishes to signal that a specific port should be used, and this facility is provided in the ADD_ADDR option as documented in Section 3.4.1. It is therefore feasible to allow multiple subflows between the same two addresses but using different port pairs, and such a facility could be used to allow load balancing within the network based on 5-tuples (e.g., some ECMP implementations [RFC2992]).

3.9.2. Delayed Subflow Start and Subflow Symmetry

Many TCP connections are short-lived and consist only of a few segments, and so the overheads of using MPTCP outweigh any benefits. A heuristic is required, therefore, to decide when to start using additional subflows in an MPTCP connection. Experimental deployments have shown that MPTCP can be applied in a range of scenarios so an implementation is likely to need to take into account factors including the type of traffic being sent and duration of session, and this information MAY be signalled by the application layer.

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

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

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

Note that this heuristic alone is probably insufficient. Traffic for many common applications, such as downloads, is highly asymmetric and the host that is multihomed may well be the client that will never fill its buffers, and thus never use MPTCP according to this heuristic. Advanced APIs that allow an application to signal its traffic requirements would aid in these decisions.

An additional time-based heuristic could be applied, opening additional subflows after a given period of time has passed. This would alleviate the above issue, and also provide resilience for low-bandwidth but long-lived applications.

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

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

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

3.9.3. Failure Handling

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

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

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

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

4. Semantic Issues

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

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

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

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

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

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

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

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

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

5. Security Considerations

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

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

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

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

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

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

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

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

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

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

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

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

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

Possible ways of improving MPTCP security could include:

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

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

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

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

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

- available flags in MP_CAPABLE (Figure 4);

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

- the version field in MP_CAPABLE (Figure 4);

6. Interactions with Middleboxes

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

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

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

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

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

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

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

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

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

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

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

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

- NATs [RFC3022] (Network Address (and Port) Translators) change the source address (and often source port) of packets. This means that a host will not know its public-facing address for signaling in MPTCP. Therefore, MPTCP permits implicit address addition via the MP_JOIN option, and the handshake mechanism ensures that connection attempts to private addresses [RFC1918], since they are authenticated, will only set up subflows to the correct hosts. Explicit address removal is undertaken by an Address ID to allow no knowledge of the source address.

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

- Traffic Normalizers [norm] may not allow holes in sequence numbers, and may cache packets and retransmit the same data. MPTCP looks like standard TCP on the wire, and will not retransmit different data on the same subflow sequence number. In the event of a retransmission, the same data will be retransmitted on the original TCP subflow even if it is additionally retransmitted at the connection level on a different subflow.

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

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

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

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

- TCP options may be removed, or packets with unknown options dropped, by many classes of middleboxes. It is intended that the initial SYN exchange, with a TCP option, will be sufficient to identify the path capabilities. If such a packet does not get through, MPTCP will end up falling back to regular TCP.

- Segmentation/Coalescing (e.g., TCP segmentation offloading) might copy options between packets and might strip some options. MPTCP's data sequence mapping includes the relative subflow sequence number instead of using the sequence number in the segment. In this way, the mapping is independent of the packets that carry it.

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

7. Acknowledgments

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

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

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

8. IANA Considerations

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

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

Table 1: TCP Option Kind Numbers

8.1. MPTCP Option Subtypes

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

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

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

Table 2: MPTCP Option Subtypes

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

8.2. MPTCP Handshake Algorithms

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

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

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

Table 3: MPTCP Handshake Algorithms

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

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

8.3. MP_TCPRST Reason Codes

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

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

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

Table 4: MPTCP MP_TCPRST Reason Codes

9. References

9.1. Normative References

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

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

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

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

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

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

9.2. Informative References

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

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

[norm] Handley, M., Paxson, V., and C. Kreibich, "Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics", Usenix Security 2001, 2001, <http://www.usenix.org/events/sec01/full_papers/handley/handley.pdf>.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Appendix A. Notes on Use of TCP Options

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

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

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

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

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

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

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

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

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

Appendix B. TCP Fast Open and MPTCP

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

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

B.1. TFO cookie request with MPTCP

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

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

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

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

B.2. Data sequence mapping under TFO

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

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

B.3. Connection establishment examples

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

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

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

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

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

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

Figure 19: The listener supports TFO

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

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

Figure 20: The listener does not support TFO

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

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

Figure 21: Partial data acknowledgement

Appendix C. Control Blocks

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

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

C.1. MPTCP Control Block

The MPTCP control block contains the following variable per connection.

C.1.1. Authentication and Metadata

Local.Token (32 bits): This is the token chosen by the local host on this MPTCP connection. The token must be unique among all established MPTCP connections, and is generated from the local key.

Local.Key (64 bits): This is the key sent by the local host on this MPTCP connection.

Remote.Token (32 bits): This is the token chosen by the remote host on this MPTCP connection, generated from the remote key.

Remote.Key (64 bits): This is the key chosen by the remote host on this MPTCP connection

MPTCP.Checksum (flag): This flag is set to true if at least one of the hosts has set the A bit in the MP_CAPABLE options exchanged during connection establishment, and is set to false otherwise. If this flag is set, the checksum must be computed in all DSS options.

C.1.2. Sending Side

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

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

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

C.1.3. Receiving Side

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

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

C.2. TCP Control Blocks

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

Note that the TCP control block on the TCP subflows does not contain the RCV.WND and SND.WND state variables as these are maintained at the MPTCP connection level and not at the subflow level.

Inside each TCP control block, the following state variables are defined.

C.2.1. Sending Side

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

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

C.2.2. Receiving Side

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

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

Appendix D. Finite State Machine

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

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

Figure 22: Finite State Machine for Connection Closure

Appendix E. Changes from RFC6824

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

- 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: Informational
Expires: September 22, 2016

R. Winter
NEC Laboratories Europe
M. Faath
University of Applied Sciences Augsburg
A. Ripke
NEC Laboratories Europe

March 21, 2016

Multipath TCP Support for Single-homed End-systems
draft-wr-mptcp-single-homed-07

Abstract

Multipath TCP relies on the existence of multiple paths between end-systems. These are typically provided by using different IP addresses obtained by different ISPs at the end-systems. While this scenario is certainly becoming increasingly a reality (e.g. mobile devices), currently most end-systems are single-homed (e.g. desktop PCs in an enterprise). It seems also likely that a lot of network sites will insist on having all traffic pass a single network element (e.g. for security reasons) before traffic is split across multiple paths. This memo therefore describes mechanisms to make multiple paths available to multipath TCP-capable end-systems that are not available directly at the end-systems but somewhere within the network.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The IETF has specified a multipath TCP (MPTCP) architecture and protocol where end-systems operate a modified standard TCP stack which allows packets of the same TCP connection to be sent via different paths to an MPTCP-capable destination ([RFC6824], [RFC6182]). Paths are defined by sets of source and destination IP addresses. Using multiple paths has a number of benefits such as an increased reliability of the transport connection and an effect known as resource pooling [resource_pooling]. Most end-systems today do not have multiple paths/interfaces available in order to make use of multipath TCP, however further within the network multiple paths are the norm rather than the exception. This memo therefore describes ways how these multiple paths in the network could potentially be made available to multipath TCP-capable hosts that are single-homed.

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

```
                               +-------+
                               | DHCP  |
     +-------+      +----------+ Server|
     |       |      |          |       |
     | Host  +------+          +-------+
     |       |      |      +-------+        ISP 1
     +-------+      +------+       |----------
                           | Gatew.|
                           |       |----------
                           +-------+        ISP 2
```

Figure 1: Reference Scenario

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

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

2. Approaches to Use Multiple Paths in the Network

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

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

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

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

```
       Host                Gateway

                         +-----------------+     ISP1
     +--------+          | src.            |
     |  virt. | 10.1.2.5 | 10.1.0.0/16  __.+----------
     |        +---+      |        __.--'   |
     |  phys. |   |      |  __.--'      N  |
     |        +----------+.:_           A  |
     |        | 10.2.2.6 |   '-.._      T  |
     +--------+          | src.   '-.._    |     ISP2
                         | 10.2.0.0/16 '-..+----------
                         |                 |
                         +-----------------+
```

Figure 2: Gateway internals

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

The way the gateway has received its public IP addresses is not relevant. It could be via DHCP, IPCP or static configuration. In order to configure the hosts behind the gateway, we propose to make use of provisioning domains [RFC7556], more specifically one provisioning domain per external gateway interface (the two interfaces to ISP1 and ISP2 in Figure 2). The DHCPv6 specification for encoding provisioning domains can be found in [I-D.ietf-mif-mpvd-dhcp-support].

In order to signal to the host, that each provisioning domain will result in a different path towards the Internet, this memo introduces a new DHCP option called EXT_ROUTE, which will be included in each provisioning domain sent by the server. The option value will determine which external interface is used to sent the traffic when using the configuration information present in the respective provisioning domain.

Upon receipt of a DHCP offer including multiple provisioning domains, or multiple offers each including one or more provisioning domains, the client SHOULD create up to n virtual interfaces, where n is one less than the number of different EXT_ROUTE option values found in all received provisioning domains. Each virtual interface will contact the DHCP server and will request configuration information for the respective provisioning domains, excluding the configuration of the physical interface.

2.2. Heuristic Use of Multiple Paths

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

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

The ADD_ADDR option defined in [RFC6824] can be used to advertise the same address but a different port to open another subflow. Additionally, the MP_JOIN option can also be used to open another subflow with the same IP address and e.g. a different source port given that a different address ID is used. This means there are multiple scenarios possible (e.g. either sender-initated or receiver-initiated) where single-homed end-hosts can influence the 5-tuple (source and destination IP addresses and port numbers plus protocol number) which is often used as the basis for per-flow load balancing. Changing the 5-tuple will only with a certain probability result in using a different path unless the load-balancing algorithm that is used is known to the MPTCP implementation (an assumption we cannot generally make). This means that a number of subflows might end up on the same path. Fortunately, the MPTCP congestion control algorithm will make sure that the collection of subflows on that path will not be more agressive than a single TPC flow.

3. Other scenarios and extensions

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

4. Alternative approaches

One alternative is that a DHCP server always sends n offers, where n is the number of interfaces at the gateway to different ISPs. The client could then accept all or a subset of these offers. This approach seems interesting in environments where there are multiple DHCP servers, one for each ISP connection (think multiple home gateways). However, accepting multiple offers based on a single DHCP request is not standard’s compliant behavior (at least for the DHCPv4 case). Also, to cater for a scenario that only contains a single DHCP server, server changes are needed in any case. Finally, correct routing is not always guaranteed in these scenarios.

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

5. Acknowledgements

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

6. IANA Considerations

One new DHCP options is required by this version of this document.

7. Security Considerations

TBD.

8. References

8.1. Normative References

[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, <http://www.rfc-editor.org/info/rfc1918>.

[RFC2991] Thaler, D. and C. Hopps, "Multipath Issues in Unicast and Multicast Next-Hop Selection", RFC 2991, DOI 10.17487/RFC2991, November 2000, <http://www.rfc-editor.org/info/rfc2991>.

[RFC7556] Anipko, D., Ed., "Multiple Provisioning Domain Architecture", RFC 7556, DOI 10.17487/RFC7556, June 2015, <http://www.rfc-editor.org/info/rfc7556>.

8.2. Informative References

[ECMP_DEPLOYMENT] Augustin, B., Friedman, T., and R. Teixeira, "Measuring Multipath Routing in the Internet", October 2011, <http://www.paris-traceroute.net/images/ton_2011.pdf>.

[I-D.ietf-mif-mpvd-dhcp-support] Krishnan, S., Korhonen, J., and S. Bhandari, "Support for multiple provisioning domains in DHCPv6", draft-ietf-mif-mpvd-dhcp-support-02 (work in progress), October 2015.

[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>.

[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>.

[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>.

[resource_pooling]
Wischik, D., Handley, M., and M. Bagnulo Braun, "The Resource Pooling Principle", October 2008, <http://ccr.sigcomm.org/online/files/p47-handleyA4.pdf>.

Authors' Addresses

Rolf Winter
NEC Laboratories Europe
Kurfuersten-Anlage 36
Heidelberg 69115
Germany

Email: rolf.winter@neclab.eu

Michael Faath
University of Applied Sciences Augsburg
An der Hochschule 1
Augsburg 86161
Germany

Email: michael.faath@hs-augsburg.de

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

Email: andreas.ripke@neclab.eu