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

B. Hesmans
O. Bonaventure
UCLouvain

July 06, 2016

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

Abstract

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

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction 2
2. Basic operation 3
3. Multipath TCP Socket API 5
 3.1. Subflow list 5
 3.2. Open subflow 7
 3.3. Close subflow 8
 3.4. Get subflow tuple 9
 3.5. Subflow socket option 9
4. IANA considerations 10
5. Security considerations 11
6. Conclusion 11
7. Acknowledgements 11
8. References 11
 8.1. Normative References 11
 8.2. Informative References 12
Authors' Addresses 13

1. Introduction

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

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

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

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

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

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

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

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

2. Basic operation

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

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

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

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

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

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

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

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

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

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

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

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

3. Multipath TCP Socket API

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

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

3.1. Subflow list

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

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

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

Figure 1: The mptcp_sub_ids and mptcp_sub_status structures

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

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

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

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

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

optlen = 42;
```

```
ids = malloc(optlen);

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

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

Figure 2: Sample code for the utilisation of MPTCP_GET_SUB_IDS

3.2. **Open subflow**

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

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

Figure 3: The mptcp_sub_tuple structure

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

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

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

An example is provided in figure Figure 4.

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

```
   int error;

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

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

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

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

   addr++;

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

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

Figure 4: Sample code to establish an additional subflow

3.3. Close subflow

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

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

Figure 5: The mptcp_close_sub_id structure

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

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

3.4. Get subflow tuple

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

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

Figure 6: The mptcp_sub_tuple structure

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

A sample code is provided in figure Figure 7.

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

optlen = 100;

sub_tuple = malloc(optlen);

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

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

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

sin++;

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

Figure 7: Sample code using the MPTCP_GET_SUB_TUPLE option

3.5. Subflow socket option

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

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

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

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

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

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

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

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

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

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

Figure 9: Example socket option

4. IANA considerations

There are no IANA considerations in this document.

5. Security considerations

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

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

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

6. Conclusion

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

7. Acknowledgements

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

8. References

8.1. Normative References

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

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

8.2. Informative References

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

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

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

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

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

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

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

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

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

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

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

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

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

Authors' Addresses

Benjamin Hesmans
UCLouvain

Email: Benjamin.Hesmans@uclouvain.be

Olivier Bonaventure
UCLouvain

Email: Olivier.Bonaventure@uclouvain.be