An enhanced socket API for Multipath TCP

Benjamin Hesmans, Olivier Bonaventure
ICTEAM, Université catholique de Louvain
Louvain-la-Neuve – Belgium
firstname.name@uclouvain.be

ABSTRACT

Multipath TCP is a TCP extension that enables hosts to send data belonging to a single TCP connection over different paths. It was designed as an incrementally deployable evolution of TCP. For this reason, the Multipath TCP specification assumes that applications use the unmodified `socket` interface. Given the growing interest in using Multipath TCP for specific applications, there is a demand for an advanced API that enables application developers to control the operation of the Multipath TCP stack. Keeping with the incremental deployment objectives of Multipath TCP, we propose a simple but powerful `socket` API that uses new socket options to control the operation of the underlying stack. We implement this extension in the reference implementation of Multipath TCP in the Linux kernel and illustrate its usefulness in several use cases.

CCS Concepts

•**Networks** → **Programming interfaces;**

Keywords

MPTCP; network; API

1. INTRODUCTION

Multipath TCP [7] is a recent TCP extension that enables hosts to send data belonging to a single connection over multiple paths. The design of this TCP extension was motivated by the desire to allow modern endhosts to efficiently utilize their different interfaces [6].

A typical example are today's smartphones that are equipped with cellular and WiFi interfaces. With regular TCP, smartphones either use their cellular interface or their WiFi interface to support applications and handovers result in the termination of the established TCP connections. Multipath TCP allows smartphones to behave differently. With Multipath TCP, a smartphone can simultaneously use its cellular and WiFi interfaces to transfer data [16, 3, 12]. Multipath TCP is able to pool the bandwidth of the WiFi and cellular interfaces [18]. This has motivated large network providers in Korea to partner with smartphone vendors to include Multipath TCP on several Android smartphones to pool the cellular and WiFi bandwidth to reach up to 800 Mbps [21]. Given that Multipath TCP is not yet deployed on regular Internet servers, these smartphones use Multipath TCP to reach a SOCKS [11] proxy which then uses regular TCP to interact with the remote servers. A similar approach is used for the OverTheBox service being rolled out by OVH [13] and there is ongoing work within the IETF to standardise a solution with Multipath TCP proxies to enable operators to efficiently combine different access networks such as xDSL and LTE [2].

Pooling bandwidth is not the only benefit of Multipath TCP on smartphones. In September 2013, less than nine months after the publication of [7], Apple enabled Multipath TCP to improve the user experience with the Siri voice recognition application on all iPhones and iPads. In this case, Apple controls both the clients and the servers and can thus use Multipath TCP in a real end-to-end deployment. The Siri application uses a long-lived TLS session and Multipath TCP provides a very fast handover when the cellular or WiFi interface becomes lossy.

Other use cases have been proposed for Multipath TCP : datacenters [17], vehicular networks [23] and various forms of Multipath TCP proxies [1].

The ongoing Multipath TCP deployments [1] have demonstrated that the protocol can be deployed in today's Internet despite the presence of various types of middleboxes [10]. Given this deployment, there is a growing interest among application developers and system designers to better control the utilisation of the different paths that can be exploited by Multipath TCP. However, as of this writing, there is no standard API that enables developers to accurately control the underlying Multipath TCP stack.

In this paper, we propose a simple and expandable extension to the `socket` API that enables applications to efficiently control the underlying Multipath TCP stack. We first briefly describe in section 2 the basic principles of Multipath TCP and the existing APIs. We describe in section 3 our proposed `socket` options. We implement[1] this API in the reference implementation of Multipath TCP in the Linux kernel [14] and use it in section 4 to illustrate its applicability to different use cases. Section 5 concludes the paper and discusses several directions for further work.

ANRW '16, July 16 2016, Berlin, Germany

ISBN 978-1-4503-4443-2/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2959424.2959433

[1]Our code will be available on https://multipath-tcp.org

2. MULTIPATH TCP

Multipath TCP [7] includes a variety of techniques to enable data belonging to one connection to be transmitted over multiple paths. Detailed information about the operation of the protocol may be found in [7, 18]. Due to space limitations, we only discuss the interactions between an application and Multipath TCP and leave the protocol details outside this paper.

From an architectural viewpoint, a Multipath TCP can be seen as a set of TCP connections, called subflows in [7], that are grouped together and managed by the two endpoints of the Multipath TCP connection. This set of subflows is not static and subflows can be established and terminated during the lifetime of a Multipath TCP connection. Multipath TCP uses various different TCP connections to exchange control information. A detailed overview of these options is outside the scope of this section, but we mention two of these that are discussed later in the text. The `ADD_ADDR` option allows to advertise the addresses assigned to a host. For example, a dual stack server will use this option to advertise its IPv6 address over the Multipath TCP connections that client have initiated over IPv4. The `REMOVE_ADDR` option is used to indicate that a previously advertised addressed does not anymore belong to the host. For example, a smartphone will send this option on its established Multipath TCP connections when it looses connectivity to its WiFi access point.

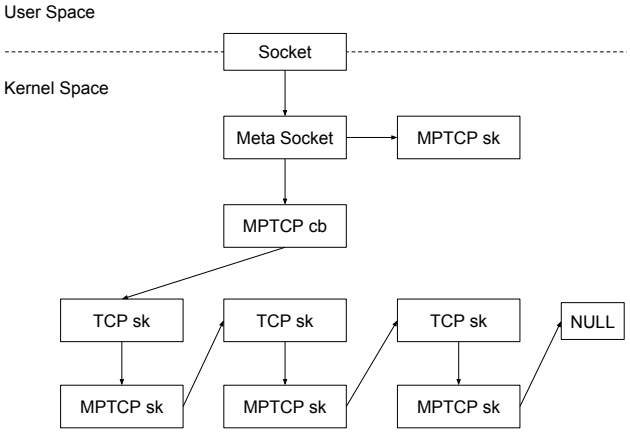

Figure 1: MPTCP socket structure

On the Linux implementation [14], when an application creates a Multipath TCP connection, two data structures are created in the kernel : a meta socket and a subflow socket. The meta socket is the only data structure that is directly linked to the socket that is visible by the application as presented on figure 1. All data sent and received on a Multipath TCP connection passes through this socket. The meta socket contains pointers various structures including the linked list of the established subflows. The sockets corresponding to these subflows are not directly visible by the application with the current Linux implementation [14]. In this implementation, the subflows are managed by a pluggable kernel module called the *path manager*. These path managers react to events such as the activation/deactivation of an interface by creating/removing the required subflows, but they do not expose an API to the application. The current Linux implementation was designed to allow an unmodified socket application to benefit from Multipath TCP without any possibility to precisely control the utilisation of the interfaces.

```
/* socket creation */
s = socket(AF_MULTIPATH, SOCK_STREAM, IPPROTO_TCP);

/* creation of first subflow */
sa_endpoints_t endpoints;
/* any source interface */
endpoints.sae_srcif = 0;
/* any address of the client */
endpoints.sae_srcaddr = NULL;
endpoints.sae_srcaddrlen = 0;
/* server address */
endpoints.sae_dstaddr = (struct sockaddr *)
                              daddr->ai_addr;
endpoints.sae_dstaddrlen = daddr->ai_addrlen;

int rc = connectx(s, &endpoints, SAE_ASSOCID_ANY,
                  0, NULL, 0, NULL, NULL);
```

Figure 2: Using Multipath TCP on iOS and OS/X

2.1 Multipath TCP APIs

To our knowledge, there are currently only three proposed APIs to control a Multipath TCP stack. The MPTCP WG of the IETF developed a minimum set of extensions to the `socket` API in [19]. However, this simple API has not yet been adopted by implementors [5].

When Apple decided to adopt Multipath TCP to support Siri, they opted for a special API to enable and use Multipath TCP. To our knowledge, this API has not been publicly documented, but several open source files released by Apple contain code examples that use Multipath TCP[2] . On iOS and OS/X, Multipath TCP is enabled by creating a socket that uses the `AF_MULTIPATH` address family. The `sa_endpoints_t` structure is used to represent the endpoints of a TCP subflow. It also contains an interface identifier. The `connectx` system call replaces the classical `connect` call. It takes as main parameters a socket and a `sa_endpoints_t` data structure that specifies the destination address of the connection and optionally the source interface and/or the source address. This system call can be used with both IPv6 and IPv4 addresses. If the server (or a middlebox on the path) does not support Multipath TCP, `connectx` returns an error and the application can convert the connection to a TCP connection by using the undocumented `peeloff` system call. If the server supports Multipath TCP, then the application can send and receive data through the normal `send/receive` system calls. An additional subflow can be attached to the connection by using again the `connectx` system call on the same socket but with a different `sa_endpoints_t` data structure. This new subflow can be created by specifying a different source interface, source address or destination address than the initial subflow. Figure 2 provides a short sample code that shows how Multipath TCP can be used by an OS/X application.

The Linux implementation [14] takes a different approach. When Multipath TCP is enabled through a `sysctl`, it replaces TCP for all applications. When an application creates a TCP socket, it is automatically promoted as a Multipath TCP socket. The establishment of the subflows is

[2]See http://opensource.apple.com/source/network_cmds/network_cmds-457/mptcp_client/mptcp_client.c or http://opensource.apple.com//source/netcat/netcat-20/netcat.c

controlled by a kernel module called the *path manager*. The *full-mesh* path manager creates a subflow between all pairs of addresses of the client and the server addresses that it has learned through the `ADD_ADDR` option. The server does not create subflows because the client could be behind a NAT or firewall. This path manager is well adapted for smartphones that interact with single-homed servers. The *ndiffports* path manager was designed for single homed clients in datacenters [17]. It creates n subflows with different source ports towards a single-homed server.

In [9], a new `netlink` *path manager* has been proposed. Instead of reacting directly to various kernel events it relays them trough a new specific `netlink` family to an user space daemon. Among others, events presented through the `netlink` socket include: the creation of a new connection and its identifier, the confirmation of the establishment of a subflow, the termination of a subflow deletion with a reason code, the availability of a new remote address, ... Since the kernel module does not react to the kernel events, the specific `netlink` family is also able to receive commands from the user space daemon to the kernel module to control each Multipath TCP connection. The set of commands usable by the daemon through `netlink` includes the creation, deletion and prioritization of subflows. In this case all the logic is ported in the user space but not directly in the application itself. The application cannot directly influence the kernel stack because the daemon handles transparently all Multipath TCP connections. It is however not excluded for an application to communicate with the daemon but no mechanism has been defined in [9] for this purpose.

3. A SOCKET API FOR MPTCP

The *path managers* used by the Linux Multipath TCP implementation implement simple behaviours that may not be adapted to all applications. To fully benefit from Multipath TCP, advanced applications should be able to query the underlying Multipath TCP stack for the following information :

- How many subflows are established ?
- What is the state of the active subflows ?

Once the application gathers enough data, it should be able to take decision and then it should be able to issue commands to the Multipath TCP stack to perform the following actions :

- Create a new subflow
- Terminate a new subflow
- Change subflows priorities
- Control subflows like an application would control simple TCP connection

In order to be usable, the API should be designed with three objectives in mind. First, it should be as compatible as possible with regular TCP so that existing applications can benefit from MPTCP without any change. This new API should not break any existing application. Second, it should provide advanced facilities that are easy to use for a developer who only knows the standard socket API. Unnecessary complex APIs are meant to be unused. Third, it should be expandable. A good API should be able to easily evolve as the developers needs change. In the rest of this section, we propose an MPTCP API that has been implemented above the existing `getsockopt` and `setsockopt` system calls. A summary of our implemented socket options is provided in table 1.

```
struct mptcp_sub_status {
    __u8  id;
    __u16 slave_sk:1,
          fully_established:1,
          attached:1,
          low_prio:1,
          pre_established:1;
};

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

Figure 3: Structure to retrieve subflow list

3.1 Querying the MPTCP socket

A simplified version of the kernel view of a socket is presented on figure 1. Underneath the meta socket visible by the application lie severals subsockets, one for each subflow. The head of the list of subflows is accessible via the `mptcp_cb` structure. The next subflow in the list is then indirectly accessible via the `mptcp_sk` structure. The application is not aware of those subsockets. To expose this information to the application, the socket option `MPTCP_GET_SUB_IDS` fills a `mptcp_sub_ids struct` (see figure 3) with a list of subflows identified by their `ID`. This `ID` is used in subsequent calls to specify the subflow on which an action is performed. Specific subflow status is present in the structure. For instance the back up status of the subflow : `low_prio`. It is possible to extend this structure to expose more subflow specific information. The `ID` used by the kernel to identify a subflow may not always link to the same subflow. The number of subflows in the Linux kernel is limited to 32 [14] and the `ID` is assigned in a range from 0 to 31 because it is mapped to a 32 bit field. It is thus impossible to have more that 32 simultaneous established subflows associated to one MPTCP connection. However, it is possible to have more than 32 non-simultaneous subflows and this has been observed on long Multipath TCP connection [4]. In this case `IDs` will be recycled by the kernel. A given `ID` can thus correspond to different subflows at different times.

3.2 Get subflow tuple

Because multiple sockets can lie behind a socket descriptor in user space, the semantics of system calls like `getsockname` and `getpeername` becomes less clear. According to [19], they should return the IP addresses of the initial subflow even if it does not exist anymore, at least for applications that are unaware of Multipath TCP. For applications that are aware of Multipath TCP, we implement the `MPTCP_GET_SUB_TUPLE` option that allows to retrieve the four tuple (source and destination addresses and ports) of a given subflow. When using `getsockopt`, the application fills the structure with the id of the subflow of interest and the kernel fills the `struct` with the pairs of addresses and ports of this subflow. This structure (called `sub_tuple` in our implementation) can later be reused to create another subflow.

Name	Input	Output	Description
MPTCP_GET_SUB_IDS	-	subflow list	Get the current list of subflows viewed by the kernel
MPTCP_GET_SUB_TUPLE	id	sub tuple	Get the pair ip and ports used by the subflow identified by id
MPTCP_OPEN_SUB_TUPLE	tuple	-	Request a new subflow with pair of ip and ports
MPTCP_CLOSE_SUB_ID	id	-	Close the subflow identified by id
MPTCP_SUB_GETSOCKOPT	id, sock opt	sock ret	Redirects the `getsockopt` given in input to the subflow identified by id and return the value returned by the operation
MPTCP_SUB_SETSOCKOPT	id, sock opt	-	Redirects the `setsockopt` given in input to the subflow identified by id.

Table 1: Implemented MPTCP socket options

3.3 Creating subflows

In the current implementation of MPTCP in the linux kernel, an application cannot directly create or terminate subflows within a connection. There are situations where an application has more information than the kernel to decide whether a new subflow should be created. For example, an application that uses HTTP could use the size of the HTTP object indicated in the response to decide whether it is useful to create another subflow. Another example is an application knowing the monetary cost associated with the utilization of each interface and only using the most expensive one when it has no alternative. Another typical use case, is an application that is not time critical and knows that one interface should not be used by default.

To create a new subflow, an application can use the `MPTCP_OPEN_SUB_TUPLE` socket option. The application must fill a `struct sub_tuple` with the pair of addresses and the destination port for the new subflow. The `ID` allocated by the kernel will be returned using the `struct` and the `setsockopt` return value indicates if the subflow has been created.

3.4 Terminating subflows

On the other hand, some applications may want to close a subflow. One of the proposed use case is the following : a security application could decide to use a specific interface to negotiated its encryption keys and use another less costly interface to exchange encrypted data. For this, it simply needs to let the kernel create the first subflow over the costly interface, negotiate the keys and then create another subflow over the cheap but less secure interace. To terminate a subflow, the application can use the `MPTCP_CLOSE_SUB_ID` socket option. This option takes as parameter the id of the subflow that must be closed.

3.5 `socket` options

Socket options play an important role in modern TCP stacks. The Linux TCP stack contains a dozen of non-standard `socket` options. These options are used by advanced applications to configure or retrieve information from the stack. In Linux, socket options can be passed at different levels, namely the socket itself, TCP or IP. We modified the `setsockopt` system call to preserve this genericity and allow an application to set an option at any level. We provide the `struct mptcp_sub_setsockopt` whose content is shown in figure 4 to encapsulate a socket option that needs to be delivered to a specific subflow.

As an illustration, figure 5 shows how the DSCP of a particular subflow can be set by an application. Our `MPTCP_SUB_SETSOCKOPT` option uses a generic structure which can encapsulate any socket option operating at any level. The application developer simply needs to specify the option name, level, and value (with the associated length) and the subflow on which is must be applied. A variant of this call allows to set the same option on all subflows of a Multipath TCP connection in a single call.

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

Figure 4: Subflow socket option structure

```
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; // subflow id
sub_sso.level = IPPROTO_IP; // option level
sub_sso.optname = IP_TOS; // socket option
sub_sso.optlen = sub_optlen;
sub_sso.optval = (char *) &val; // value

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

Figure 5: Passing a `socket` option to a specific subflow

Another example is the `TCP_INFO` socket option. This Linux specific socket option can be used to retrieve the state of the underlying TCP connection and various statistics including the current value of the congestion window, the number of bytes transmitted, the retransmission timer, ... Some applications use this socket option to detect underperforming TCP connections. With our enhanced API, they could query all the subflows associated to a give Multipath TCP connection to detect the underperforming ones and possibly terminate them.

4. USE CASES

We illustrate the flexibility of our enhanced `socket` API by using it to implement several applications that demonstrate how the Multipath TCP stack can be controlled by an application.

```
rotate_tresh = X
rotate_count = 1
tot_bytes    = 0

while (n = read()) != 0 do
  tot_bytes += n
  if tot_bytes > rotate_count * rotate_tresh then
    refresh_sub()
    rotate_count ++
```

Figure 6: Subflow refresh algorithm

4.1 Refreshing subflows

Our first example is an application that replaces the current subflow with a new one after having received `x` bytes of data. This type of control could be motivated by security (e.g. an `ssh` session moves from one interface to another to make interception more difficult) or performance reasons (e.g. a mobile network uses a rate limiter that slows down long TCP connections). We modify `curl` and include the very simple algorithm shown in figure 6 to create new subflows. This modification requires 96 lines of code in our test application.

Our implementation leverages our new socket options. In particular, we use `MPTCP_GET_SUB_IDS` to retrieve the `id` of the active subflow and `MPTCP_GET_SUB_TUPLE` to retrieve its corresponding tuple. Then we use `MPTCP_OPEN_SUB_TUPLE` to create a new subflow. Once this subflow is active, we use `MPTCP_CLOSE_SUB_ID` to close the previous one.

In our test application we download a file over HTTP and use 1024 for `rotate_tresh`. Our application creates up to 56 successive subflows to download the file.

4.2 Delayed subflow establishment

Our second example is motivated by the fact that measurement studies have shown that, with the default path manager, the Linux Multipath TCP implementation often creates subflows that do not transfer any data [4]. A typical example are the short HTTP requests. We modify `curl` to parse the `Content-Length` header and only create a second subflow if the HTTP object is larger that a given threshold. This modification required only 10 lines of code. If this small change was deployed on Multipath TCP enabled smartphones, it could significantly reduce the number of established subflows [4].

A second example is a streaming application. These applications work provided that data is received at a regular rate. Many video streaming applications operate over HTTP and download chunks of video frames at a regular rate. To display the video to the user, it needs to receive the chunks at a regular rate. We modify `curl` to simulate such a streaming application and set a deadline for each requested HTTP object. If the application knows the acceptable delay d to download a chunk of data of size s, it checks after $d/2$ that at least half of the data have been received. If it is not the case then it decides to open a new subflow over a secondary interface to speed up the download. A more generic variant of this solution could be to check more regularly if the download rate is sufficient to achieve the download in the acceptable delay for the application. If the application has a delay d to download a file of size s. If the download started at t_0, at any point in time t_n, given the size of the chunk already download s_n, if $s_n/(t_n - t_0) < s/d$ then a new subflow should be open.

5. DISCUSSION

Different solutions have been proposed to enable applications to control a multipath stack. `SCTP` can be controlled through an enhanced socket API but also control messages and other system calls [22]. For Multipath TCP, two approaches described in section 2 have already been implemented. Apple opted for a new type of Multipath TCP `socket` with dedicated system calls. The SMAPP path manager proposed in [9] relies on a new `netlink` family to expose the path manager functions to user space. With this approach a userspace daemon can control the entire operation of the underlying stack. We now compare these three approaches.

Our first point of comparison, is the simplicity of the approach from a developer's viewpoint. New system calls and the Netlink solution force the developers to learn a way to organise their code. This will force the developer to maintain two different versions of the application : one using regular TCP and one supporting Multipath TCP. The Netlink solution is more transparent to the developer since the user space daemon can take a decision on behalf of the application without any change, agreement or communication with it. However, the daemon needs to be adapted every time a new behaviour is requested for a given application. Our enhanced socket is less transparent to the developer, but most developer are already used to using socket options and the change is very limited.

A second point of comparison is the level on control brought by the solution to the application. The system call solution brings a really fine control over the stack. The Netlink solution brings also a lot of control, but this control is kept at the daemon level, and the link between the daemon and the application remains to be done. On the other hand, our enhanced socket options provide a level of control that should be sufficient for many applications. A disadvantage of our socket API compared to the Netlink solution is that is does not currently allow the application to directly react to specific events such as the failure of one interface or the expiration of retransmission timers.

A third point of comparison is the evolvability of the solution. System calls are not really meant to evolve rapidly because each new system call must be learned by the application developer. From the Netlink perspective, new event/commands can be introduced easily but new bindings between the daemon and the applications need to be created. Our socket option uses an approach that is already well know by most developers and new socket options can be easily introduced if needed.

Our proposed socket options are well adapted to send command from user space to kernel space and to retrieve information of the kernel state into user space. However it is not well suited to deliver events to an application. There are some situations where an application could want to receive events from the underlying stack instead of regularly polling it. Our further work will be to enhance the `sendmsg(2)` and `recvmsg(2)` system calls and use the ancillary data to exchange information about events in the stack. Examples of events include: a new interface is available, a subflow has been destroyed before the end of the connection, a retransmission timer has expired, ...

Another point that still requires our attention is the handling of the address advertisements. The Linux implementation handles the `ADD_ADDR` messages inside the path man-

ager module. Some path managers may store all received `ADD_ADDR` messages while others could react to the reception of such messages by creating a subflow without storing it. Unfortunately, there is currently no generic API to retrieve the list of addresses received through `ADD_ADDR` messages. One solution could be to remove the storage responsibility from the path manager module and make it a generic feature. Another solution would be to use control message to pass the information to the application and to not rely on the storage of address advertisements inside the kernel.

We expect that our enhanced socket API will be very useful to various users of Multipath TCP. Developers of native Multipath TCP running on smartphones could use it to tune their applications. The SOCKS clients and servers that are deployed in various countries [21, 13] would also benefit from this API. It could also be used to implement richer APIs like socket intents [20, 8]. Our implementation confirms that this API can be implemented in a Multipath TCP stack and we have initiated discussions with the designers of the other Multipath TCP implementations to propose a common enhanced socket API within the IETF.

6. ACKNOWLEDGMENT

This work was partially supported under the Cisco University Research Program Fund. We would like to thank Quentin De Coninck for his participation during the discussions that led to this paper.

7. REFERENCES

[1] O. Bonaventure, C. Paasch, and G. Detal. Experience with Multipath TCP. Internet-Draft draft-ietf-mptcp-experience-01, IETF Secretariat, Mar. 2015. I-D Exists.

[2] M. Boucadair et al. An MPTCP Option for Network-Assisted MPTCP Deployments: Plain Transport Mode. Internet draft, draft-boucadair-mptcp-plain-mode-07, work in progress, May 2016.

[3] Y.-C. Chen, Y.-s. Lim, R. J. Gibbens, E. M. Nahum, R. Khalili, and D. Towsley. A measurement-based study of MultiPath TCP performance over wireless networks. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 455–468, New York, NY, USA, 2013. ACM.

[4] Q. De Coninck, M. Baerts, B. Hesmans, and O. Bonaventure. A first analysis of multipath tcp on smartphones. In *Passive and Active Measurement*, pages 57–69. Springer, 2016.

[5] P. Eardley. Survey of MPTCP Implementations. Internet-Draft draft-eardley-mptcp-implementations-survey-02, IETF Secretariat, July 2013.

[6] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. Architectural Guidelines for Multipath TCP Development. RFC 6182 (Informational), Mar. 2011.

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

[8] K.-J. Grinnemo, T. Jones, G. Fairhurst, D. Ros, A. Brunstrom, and P. Hurtig. Towards a flexible internet transport layer architecture. To appear in IEEE LANMAN 2016, Rome, June 2016.

[9] B. Hesmans, G. Detal, S. Barré, R. Bauduin, and O. Bonaventure. Smapp: Towards smart multipath tcp-enabled applications. In *CoNEXT'15*, 2015.

[10] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 181–194, New York, NY, USA, 2011. ACM.

[11] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. SOCKS Protocol Version 5. RFC 1928 (Proposed Standard), Mar. 1996.

[12] Y.-s. Lim, Y.-C. Chen, E. M. Nahum, D. Towsley, and R. J. Gibbens. How green is Multipath TCP for mobile devices? In *Proceedings of the 4th Workshop on All Things Cellular: Operations, Applications, & Challenges*, pages 3–8. ACM, 2014.

[13] OVH. Overthebox. https://www.ovhtelecom.fr/overthebox/, 2016.

[14] C. Paasch, S. Barre, et al. Multipath TCP in the Linux Kernel. available from http://www.multipath-tcp.org.

[15] C. Paasch and O. Bonaventure. Multipath TCP. *Commun. ACM*, 57(4):51–57, Apr. 2014.

[16] C. Paasch, G. Detal, F. Duchene, C. Raiciu, and O. Bonaventure. Exploring Mobile/WiFi Handover with Multipath TCP. In *ACM SIGCOMM CellNet workshop*, pages 31–36, 2012.

[17] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *ACM SIGCOMM 2011*, 2011.

[18] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How hard can it be? Designing and implementing a deployable Multipath TCP. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[19] M. Scharf and A. Ford. Multipath TCP (MPTCP) Application Interface Considerations. RFC 6897 (Informational), Mar. 2013.

[20] P. S. Schmidt, T. Enghardt, R. Khalili, and A. Feldmann. Socket intents: Leveraging application awareness for multi-access connectivity. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 295–300, New York, NY, USA, 2013. ACM.

[21] S. Seo. KT's GiGA LTE. Presentation at IETF'93, see https://www.ietf.org/proceedings/93/slides/slides-93-mptcp-3.pdf, July 2015.

[22] R. Stewart, M. Tuexen, K. Poon, P. Lei, and V. Yasevich. Sockets API Extensions for the Stream Control Transmission Protocol (SCTP). RFC 6458 (Informational), Dec. 2011.

[23] N. Williams, P. Abeysekera, N. Dyer, H. Vu, and G. Armitage. Multipath TCP in Vehicular to Infrastructure Communications. Technical Report 140828A, CAIA, Swinburne University of Technology, August 2014.