SMAPP : Towards Smart Multipath TCP-enabled APPlications

B. Hesmans *, G. Detal †, S. Barre †, R. Bauduin *, O. Bonaventure *

* Université catholique de Louvain, Louvain-la-Neuve, Belgium † Tessares, Louvain-la-Neuve, Belgium
* firstname.lastname@uclouvain.be † firstname.lastname@tessares.net

ABSTRACT

Multipath TCP was designed and implemented as a backward compatible replacement for TCP. For this reason, it exposes the standard `socket` API to the applications that cannot control the utilisation of the different paths. This is a key feature for applications that are unaware of the multipath nature of the network. On the contrary, this is a limitation for applications that could benefit from specific knowledge to use multiple paths in a way that fits their needs. As the specific knowledge of an application can not be known in advance, we propose a Multipath TCP path manager that delegates the management of the paths to the applications. This path manager enables applications to control how the different paths are used to transfer data. We implement this path manager above the Linux Multipath TCP kernel. It is composed of a kernel part that exposes events and commands to an userspace application that controls the key functions of Multipath TCP such as the creation/suppression of subflows or reactions to retransmissions. We demonstrate the benefits of this path manager on different use cases.

CCS Concepts

•**Networks** → **Transport protocols;** *Programming interfaces; Network control algorithms; Network experimentation;* Network mobility;

1. INTRODUCTION

The Transmission Control Protocol (TCP) is one of the key protocols in today's Internet. It provides a reliable bytestream service and is used by a wide range of applications running on a wide range of devices ranging from smartphones to datacenters. Despite or maybe due to its popularity, TCP continues to evolve. Multipath TCP [7, 19] is one of the most recent TCP extensions. It completely changes one of the basic design assumptions of the original TCP specification : a TCP connection is always identified by a four-tuple (source and destination IP addresses and ports). All the packets that are sent for such a connection always contain this four-tuple. An annoying consequence of this coupling is that on a multihomed host, e.g. a smartphone with a cellular and a WiFi interface or a simple dual-stack PC having one IPv4 address and one IPv6 address, it is impossible to send the packets that belong to one TCP connection using a different interface/address, even if the primary one fails.

Multipath TCP [7] solves this problem by allowing the packets that belong to one connection to be sent from different interfaces/addresses. The development of Multipath TCP has been motivated by several use cases ranging from mobile devices such as smartphones that are equipped with cellular and WiFi interfaces [15], to large datacenters [18]. While previous TCP extensions like Selective Acknowledgements or the Timestamp and Window scale options took years to be widely deployed [10], Multipath TCP has been quickly adopted by one major operating system. Since September 2013, all Apple smartphones and tablets use Multipath TCP to support the Siri voice recognition application. In July 2015, Korean Telecom announced a commercial deployment of Multipath TCP on several types of Android smartphones to combine fast LTE and fast WiFi. In September 2015, OVH announced a bonding service that uses Multipath TCP to enable SMEs to bond several DSL lines together.

Multipath TCP was designed with three main compatibility objectives in mind [6]. The first is that Multipath TCP must be usable by existing applications through the existing `socket` API. The Multipath TCP implementation in the Linux kernel [14] meets this objective by allowing any application to use Multipath TCP. The second objective is that Multipath TCP must

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CoNEXT '15 December 01-04, 2015, Heidelberg, Germany
© 2015 ACM. ISBN 78-1-4503-3412-9/15/12...$15.00
DOI: 10.1145/2716281.2836113

be compatible with the deployed networks. To meet this objective, the Multipath TCP designers had to include in the protocol various mechanisms that enable it to deal with existing middleboxes [7, 19]. The third objective was the fairness with other users of the network. To meet this objective, several congestion control schemes have been proposed and implemented.

In this paper, we revisit the first compatibility objective of the Multipath TCP working group. Instead of assuming that the application is dumb and only requires a regular byte stream service, we start with the assumption that the application is adaptative and wants to obtain the best results with Multipath TCP. We expect that such smart applications will be initially implemented on smartphones and perhaps also in datacenters.

This paper is organised as follows. We first briefly present Multipath TCP and the related work in Section 2. We then propose to separate the Multipath TCP *control* and *data* planes. The *data* plane, i.e., all the functions concerned by the data transfert, remains in the kernel while we propose to move the *control* plane functions, i.e., the management of the different paths, into userspace. We then illustrate the benefits of this approach in Section 4 with four different use cases.

2. MULTIPATH TCP

Multipath TCP [7] enables hosts to exchange packets belonging to a single connection over several interfaces/paths. For this, each Multipath TCP connection is composed of several TCP connections that are called subflows [7]. As an illustration of the operation of Multipath TCP, let us consider the simple but important case of a smartphone equipped with a cellular and a WiFi interface. To create a connection with a server, the smartphone sends a `SYN` segment over the WiFi interface. This segment contains the `MP_CAPABLE` option that requests the utilisation of Multipath TCP and includes a random key. The server replies with a `SYN+ACK` segment that also contains the `MP_CAPABLE` option and a random key. The smartphone finalises the three-way handshake and the Multipath TCP connection is established. At this point, the connection is composed of only one subflow, the one established over the WiFi interface. Data can be sent over this subflow. Multipath TCP uses two levels of sequence numbers : the regular sequence number in the TCP header that tracks the bytes sent over this subflow and the data sequence number that is placed in the `DSS` option and tracks the bytes transported over the entire Multipath TCP connection. To use the cellular interface, the smartphone simply sends a `SYN` segment with the `MP_JOIN` option over this interface. This option includes a token derived from the random key exchanged in the `MP_CAPABLE` option to identify the Multipath TCP connection to which the subflow must be associated. The server confirms the establishment of the subflow with a `SYN+ACK` segment containing the `MP_JOIN` option. The smartphone finalises the three-way handshake with an `ACK` segment. At this point, the smartphone and the servers can send data over the WiFi or the cellular subflow. The Multipath TCP implementation uses a packet scheduler to decide over which available subflow each data is transmitted. Several schedulers have been implemented [16] and the default one prefers the subflow with the lowest round-trip-time provided that its congestion window is open. If one of the active subflows fails, e.g., due to a loss of connectivity, data that was initially transmitted over one subflow can be resent over the other. Due to space limitations, we focus our discussion of the related work on the articles that are directly related to the management of the subflows.

An important part of a Multipath TCP implementation is the strategy that it uses to create subflows. The Linux kernel implementation [14] includes modules that implement these strategies. As of this writing, the Linux implementation contains two strategies that are called path managers for historical reasons : `full-mesh` and `ndiffports`. In both cases, only the client creates the subflows. The server never creates subflows because the client is often behind a NAT or firewall that blocks connection attempts [5]. The `full-mesh` path manager listens to events from the underlying network interfaces and creates one subflow towards the server over each active interface. These subflows are created immediately after the creation of the connection or when an interface becomes active. It enables smartphones to react to losses of connectivity [15]. The `ndiffports` path manager creates n subflows over the same interface as the initial one immediately after the establishment of the connection. This path manager was designed for datacenters [18] where it enables the utilisation of paths that are load-balanced with Equal Cost Multipath (ECMP).

A few researchers have explored how Multipath TCP should manage the available subflows and interfaces. RFC6897 proposes some extensions to the basic socket API to enable applications to add/remove addresses to a Multipath TCP connection [21]. However, none of the existing Multipath TCP implementations implement this proposed extension [5]. Paasch et al. [15] evaluate how wireless devices can adapt to losses of connectivity. This paper proposes three modes of operation for Multipath TCP on smartphones: single-path, backup and full-mptcp. Bocassi et al. [1] propose the Binder path manager that leverages loose source routing and Multipath TCP to aggregate different paths in wireless mesh networks. Lim et al. [23] propose an extension to Multipath TCP that enables to adapt the utilisation of the subflows based on information extracted from the MAC layer. This extension is evaluated experimentally, but there are no details on how it has been implemented. Several researchers have evaluated the energy impact of using Multipath TCP on smartphones [17, 12]. Peng et al. propose models that demonstrate the benefits of managing the subflows to reduce energy consumption

but do not propose any implementation [17]. Lim et al. propose an energy-aware Multipath TCP (eMPTCP) [12]. eMPTCP delays the establishment of subflows on smartphones over the LTE interface. However, when the smartphone switches from LTE to WiFi, they propose to reset the round-trip-time estimation of the LTE subflow to zero msec to force the utilisation of this subflow [12]. This speeds up the utilisation of the LTE subflow, but is not an architecturally clean solution to the subflow management problem.

Schmidt et al. proposed the utilisation of *socket intents* [22] to allow applications to inform the stack of what they know about the future communication pattern. These *intents* include information such as the type of transfer (query, bulk, stream) or the information about the flow (number of bytes, duration, ...). We also use this kind of information in our design and *socket intents* could be a way to exchange it with the subflow controller.

3. THE SUBFLOW CONTROLLER

The Linux implementation of Multipath TCP [14] resides entirely in the kernel. Most of the kernel code is devoted to the transmission and reception of data, but the management of the subflows is also performed in the kernel by the `full-mesh` and `ndiffports` path managers. This design choice was motivated by performance reasons. An unfortunate consequence of this choice is that if an application wants to control the utilisation of the subflows, it must include a new kernel module. This is not a good solution and only three path managers have been implemented in the kernel in several years.

We reconsider this design choice by clearly separating the Multipath TCP *data* and *control* planes. The data plane includes all functions that deal with the transmission of data. It remains in the kernel for performance reasons. The control plane includes all the functions that manage the subflows that compose a Multipath TCP connection. From a performance viewpoint, there is no reason to place these functions in the kernel. Furthermore, some applications might want to implement complex policies to manage their subflows. This kind of code does not really fit inside a kernel. To enable the applications to interact with the Multipath TCP kernel code, we define a new `Netlink` family. `Netlink` [20] is an interprocess communication mechanism supported by the Linux kernel that allows applications to interact with the kernel through messages. This is similar to the approach proposed earlier by M. Coudron [2].

However, writing code to send and receive `Netlink` events can be complex for application developpers. To ease the development of subflow controllers, we abstract all the complexity of handling `Netlink` in a library [3] that is linked with the subflow controller running entirely into userspace. This library (Figure 1) interacts with the `Netlink` path manager that is part of the kernel. This path manager uses the existing in-kernel path manager interface (shown in red in Figure 1) and exposes this interface through `Netlink`. The path manager is implemented in 1100 lines of C code while the library contains 1900 lines of code.

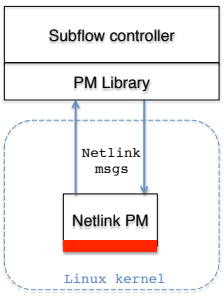


Figure 1: The subflow controller and the `Netlink` path manager

The `Netlink` path manager provides a flexible API that exposes events and state information from the kernel [3]. Callback functions provided by the subflow controller are triggered when a specific event happens in the Multipath TCP kernel or based on other inputs. The subflow controller receives only notifications for events it registered to.

The `Netlink` path manager sends and receives messages that contain information about the connection, the subflow(s), the type of event, etc. It supports many more events and commands than M. Coudron's earlier prototype [2]. The `created` event is triggered when a Multipath TCP connection is established. It contains the four-tuple, the id of the initial subflow and other information required to identify the connection. The `estab` event indicates the success of the three-way handshake and the `closed` event marks the termination of the Multipath TCP connection. These events enable a path manager to manage the connections established by several applications.

The `add_addr` and `rem_addr` events provide the IP addresses announced and removed by the remote host with Multipath TCP options [7]. Thanks to these events, the subflow controller can store the addresses of the remote host and establish new subflows only when and if needed. This is more flexible than the existing in-kernel path managers that immediately create subflows. This also reduces the state maintained for each Multipath TCP connection in the kernel.

The `sub_estab` and `sub_closed` events enable an application to control the utilisation of the subflows. The `sub_estab` event is triggered once a new subflow has been established. A server could use this event to limit the number of subflows that it currently accepts (e.g., only accept subflows originating from different addresses to prevent ressource abuse with parallel subflows). The `sub_closed` event is triggered when a `RST` segment is received over one subflow or when a subflow is terminated due to excessive retransmissions. This

event is also associated with an error code (based on standard `errno`) that indicates the reason for the removal (e.g., excessive expirations of the rto, destination unreachable, etc.).

The last event is the `timeout` event. On a TCP connection, the expiration of the retransmission timer is usually an indication of severe losses. With regular TCP, there is nothing that the application could do if the retransmission timer expires too often. With Multipath TCP, the situation is different since a second path could have very different loss characteristics than the current one. This event reports the current value of the retransmission timer and can be used as a trigger to create an additional subflow.

In addition, the `Netlink` path manager also gathers the events that are triggered by the interfaces when a local IP address is enabled (`new_local_addr`) or disabled (`del_local_addr`).

In addition to subscribing to some of these events, the library enables the subflow controller to modify the state of Multipath TCP connections through commands. Our initial implementation supports several types of commands. First, it is possible to request the creation of a subflow. A controller can create a subflow based on an arbitrary 4-tuple (IP addresses and ports). A similar command allows to remove any established subflow (either created through the controller or established by the peer). This enables the subflow controller to easily adjust the utilisation of the subflows. The controller can also retrieve information from the control block of the Multipath TCP connection or one of the subflows. In practice, this is equivalent to the utilisation of the `TCP_INFO` socket option on Linux.

4. SAMPLE USE CASES

In this section, we illustrate the benefits of the userspace subflow controller with different use cases that demonstrate how a smart application can intelligently interact with Multipath TCP.

4.1 Smarter long-lived connections

Some applications such as `ssh`, various chat applications, or notification applications on smartphones use long-lived connections that can last hours or days. These connections pose operational problems in networks that contain middleboxes like firewalls or NAT that maintain state for each established connection. The typical example is a connection that has been established but did not recently transmit data. Many NATs or firewalls will drop the state for this connection after some time. Although the IETF recommends a timeout of not less than two hours and four minutes, many deployed NATs and firewalls are more aggressive and remove unused state after a few hundreds of seconds [9]. Furthermore, many networks include cascades of such middleboxes [13]. Some applications react by sending data on a regular basis over each established connection. As an example, RFC3948 [11] recommends to send keepalive packets every 20 seconds. An unfortunate consequence of this battle between applications and middleboxes is that mobile devices need to consume a lot of energy simply to preserve the state for the established TCP connections in the middleboxes. Given the importance of energy consumption on such devices [17, 12], this is not a good approach.

Our first subflow controller is a reimplementation of the `fullmesh` path manager that is present in the Multipath TCP Linux kernel. This controller is implemented in about 800 lines of user space C code. It implements a listener for all the events described in Section 3. It listens to the `new_local_addr` and `del_local_addr` events to react to the activation and deactivation of local addresses like the in-kernel path manager. In addition, it also listens to the `sub_closed` event to react to the failure of any subflow. When such an event occurs, the subflow controller analyses the error condition (excessive timeout, `RST`, reception of an `ICMP` message, etc.) and reacts accordingly. It tries to reestablish the failed subflow and sets different timeouts based on the error condition (e.g. a short timeout if a `RST` was received and a longer timeout upon reception of an ICMP network unreachable message). Experiments with this controller show that it correctly maintains the subflows established over the different paths even under difficult network conditions.

4.2 Smarter backup

Multipath TCP [7] supports backup subflows. The backup status associated to a subflow is a binary flag that is exchanged in the `SYN` segment at subflow establishment time. It can also be changed dynamically with the `MP_PRIO` option [7]. According to RFC6824 [7], a backup subflow is a subflow that should only be used to transmit data once all other (non-backup) subflows have failed. This is the classical definition of a backup interface that works well on hosts. When an interface fails on such hosts, Multipath TCP immediately detects the failure and moves the traffic to the backup interface [15].

On mobile devices such as smartphones, the availability of one interface cannot be represented as a binary variable. When a smartphone moves around an access point or a cellular tower, there are regions where the wireless network does not work at all, regions where it works perfectly and regions where an IP address is assigned to the smartphone, but the radio conditions are so bad that most of the packets are lost. We use a Mininet [8] emulation to illustrate the situation experienced on smartphones. A connection starts over one interface and the second is set as a backup interface. After 1 second, the packet loss ratio over the primary path increases to 30%. Multipath TCP tries to retransmit the data over this interface and applies the exponential backoff to its retransmission timer until it reaches the maximum value (15 doublings on Linux). At this

point (after 12 minutes in our experiment with the default Linux configuration), TCP eventually terminates the subflow. This triggers Multipath TCP to use the backup subflow since it is the only available one.

The userspace subflow controller enables a different model for backup subflows that improves user experience. Since Multipath TCP supports break-before-make [7], our controller does not immediately establish the backup subflow. On a smartphone where the cellular interface would likely be used as a backup, this reduces both energy and radio resource consumption. The controller simply listens to the `timeout` event. When a retransmission timer expires, it checks the current value of the timer. If the timer becomes larger than a configured threshold, the subflow is considered to be underperforming. The controller then closes the underperforming subflow and creates a subflow over the backup interface to continue the transfer. This is illustrated in Figure 2a which shows the evolution of the data sequence numbers (the color indicates the subflow used to send the data). The transmission starts over the primary subflow (in green in Figure 2a). When the retransmission timer reaches one second, this subflow is terminated and a new subflow is created over the backup path (in red in Figure 2a).

4.3 Smarter streaming

We consider a simple streaming application that sends one 64 KBytes block every second. It expects that each block of data will be delivered within one second. We use this application over an emulated network with two 5 Mbps links between the client and the server. Each link has a 10 msec delay. The link bandwidth is almost an order of magnitude larger than the application goodput (520 Kbps).

For this Mininet experiment, we first use the default `full-mesh` path manager. When there is no loss, each block of 64 KBytes is delivered within 100 msec. However, when there are losses over the initial subflow the block delay quickly increases as shown by the CDF in Figure 2b.

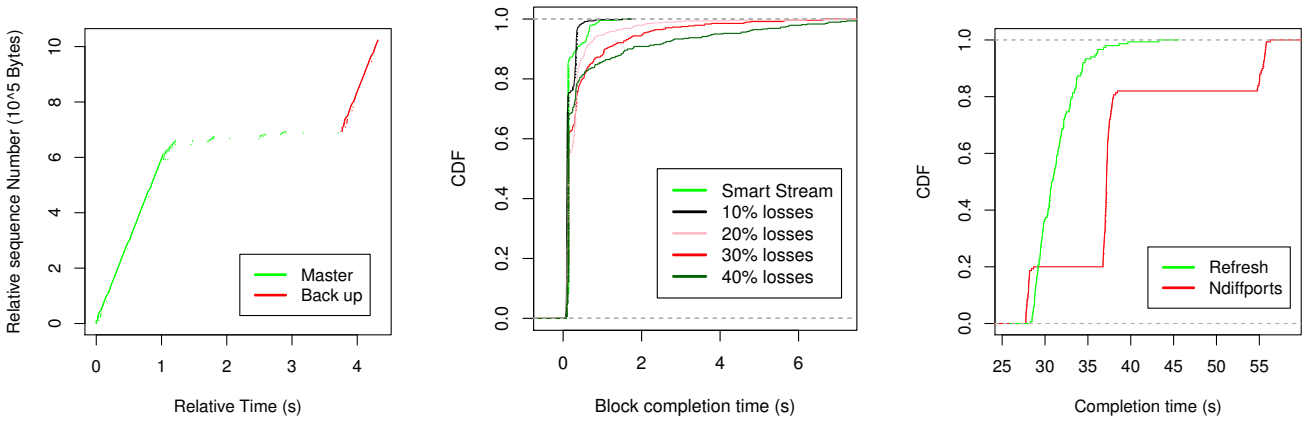
A closer look at the packet traces reveals the reasons for the low performance achieved with the default `full-mesh` path manager. When a retransmission timer expires on the initial subflow, the corresponding data can be reinjected on the other subflow. However, the data is still retransmitted on the initial subflow. If the retransmission is lost, the retransmission timer is doubled. This can happen several times and most of the data is sent on the second subflow. If at this point the scheduler decides to send some data on the underperforming subflow, this data is protected by an already very long RTO. If the data is lost, Multipath TCP waits a long time to retransmit it, which explains the long tail of the CDF in Figure 2b.

We prototype a subflow controller that expects the blocks of data to be delivered within 1 second. 500 msec after each start of block, it measures the progress of the data transfer by extracting the `snd_una` state variable from the kernel. If fewer than 32 KBytes have been sent, it considers the subflow to be underperforming and opens another subflow on the other interface. This controller also monitors the evolution of the RTO. If the RTO of a subflow becomes larger than 1 second, it is immediately closed. With this controller, if the initial subflow is fast enough to support the stream no additional subflow is established. If the initial subflow does not have enough bandwidth, a second subflow is established. The controller can also close the initial subflow if its performance is too bad. Our experiments with different packet loss ratio (not shown in the figure) for graphical reasons show that our controller provides almost the same CDF of the block delays for packet loss ratios in the 10-40% range.

4.4 Smarter exploitation of flow-based LB

In many networks, there are multiple paths between a pair of single-homed hosts given the widespread usage of Equal Cost Multipath (ECMP), link bonding and other techniques that perform flow-level load-balancing. Typically, load-balancing routers compute a hash over the four-tuple to select the path for each flow. This implies that hosts cannot easily predict which path will be used for a particular flow. The `ndiffports` kernel path manager was designed with this use case in mind [18]. If many paths are available, the n subflows that it creates are likely to use different paths. However, if the number of available paths is close to n, several subflows might use the same path.

The flexibility of our subflow controller enables a different approach to the management of the subflows in such a scenario. When the connection starts, our controller creates n subflows. These subflows use random source ports and are load-balanced in the network. Regularly (every 2.5 seconds in our current implementation), the controller queries the Multipath TCP stack to retrieve the `pacing_rate` of each subflow. This `pacing_rate` is a state variable that measures the current rate of a given TCP connection. It is included in recent versions of the Linux TCP stack [4]. Our controller compares the `pacing_rate` of the different subflows, removes the one with the lowest rate and immediately creates a new subflow. This is a very simple heuristic that will need to be updated based on experience in real networks. This controller is implemented in only 230 lines of C code. We evaluate it in a simple Mininet environment. The client and the server are attached to different routers. The two routers load-balance the flows over four available paths that have a capacity of 8 Mbps and delays of respectively 10 msec, 20 msec, 30 msec and 40 msec. The client sends a 100 MBytes file and opens 5 subflows. We expect that by opening 5 subflows over 4 paths, our controller will end up continually using all four paths, while the in-kernel `ndiffports` path manager will likely have at least some of its 5 subflows using the same path. This is illustrated



(a) The subflow controller detects when the retransmission timer becomes too long and creates the backup subflow at this time.

(b) CDF of the delay required to deliver a 64 KBytes to the client under different packet loss conditions.

(c) By regularly restablishing low-performing subflows, our subflow controller improves network utilisation

Figure 2: Results of the Mininet experiments

in Figure 2c which shows the CDF of the file transfer times. With the in-kernel `ndiffports` path manager, we can identify 3 clusters around 28 s, 37 s and 55 s, corresponding to the subflows using 4, 3 and 2 paths respectively. Even with a very simple implementation, our subflow controller tends to use the 4 available paths, outperforming the in-kernel `ndiffports` path manager significantly. For reference, the shortest time using the four paths is 27.8 s, and the worst time using only one path is 111.7 s.

4.5 User space path manager performances

As a first evaluation of the CPU cost of the user space path manager, we perform a simple experiment in a lab between two hosts connected with a direct 1 Gbps Ethernet link. The server has an Intel(R) Xeon(R) CPU X3440 @ 2.53GH and 8GB of memory. The client has an Intel(R) Core(TM)2 Duo CPU E6550 @ 2.33GHz and 4GB of memory. The server runs the `lighttpd` HTTP web server and the default in-kernel path manager. The client performs one thousand consecutive HTTP/1.0 GET queries for a 512 KB file. This experiment is performed with two variants of the `ndiffports` path manager : user space and in-kernel. These two path managers create a second subflow as soon as the initial subflow has been established. We measure the delay between the `SYN` of the initial subflow (i.e., containing the `MP_CAPABLE` option) and the `SYN` of the second subflow (i.e., containing the `MP_JOIN` option. Figure 3 provides the CDF of the delays measured with the two different path managers. It shows that the in-kernel path manager is slightly faster than the user space one. On average, the user space path manager increases the delay by 23 microseconds. This additional delay is small and remains acceptable for a client. We also performed experiments during which we stressed the CPU on the client by running additional processes. In this case, both the in-kernel and the user space path managers are affected. The delay increase due to the user space path managers remains smaller than 37 microseconds.

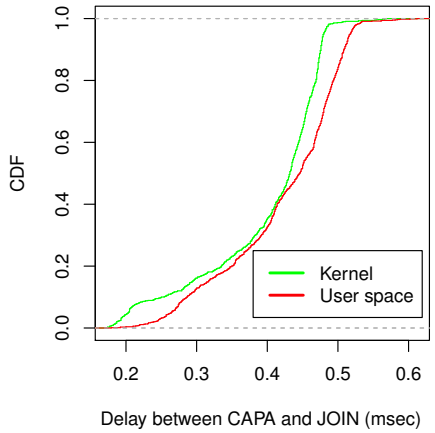


Figure 3: Kernel path manager is slightly faster than user space path manger to open a second subflow

5. DISCUSSION

Multipath TCP was designed under the assumption that applications should not be modified to use it. For this reason, it exposes the unmodified socket interface. This design choice was key to ease the deployment of Multipath TCP. Now that this deployment has started, Multipath TCP users and developers seek ways to better exploit its multipath capabilities. Our proposed subflow controller solves this problem without forcing the kernel to maintain a lot of state and handle complex policies. Our design is a first step to separate the *control* and *data* planes of Multipath TCP. The *control* plane includes all the functions related to the management of the subflows while the *data* plane includes all the functions that transmit and receive data. This design still assumes that the main parts of the networking

stack are implemented in the kernel. From an implementation viewpoint, this enables us to leverage the optimised TCP and Multipath TCP implementations in the Linux kernel. However, it also means that application developpers depend on the evolution of the kernel to use new Multipath TCP features. An alternative would be to move all the networking stack in user space as proposed by various researchers [10, 24]. This would simplify the deployment of new features and protocol extensions, but could result in the proliferation of incompatible extensions if each application developper implements his own stack. In the long-term, we expect that the success of our approach and also of the user space networking stacks will depend on the availability of well maintained high-performance librairies that can be easily used by application developpers.

Acknowledgements

This work was supported by the FP7 Trilogy2 project and the RICE project.

6. REFERENCES

[1] L. Boccassi, et al. Binder: A system to aggregate multiple internet gateways in community networks. In *Proceedings of the 2013 ACM MobiCom Workshop on Lowest Cost Denominator Networking for Universal Access*, LCDNet '13, pages 3–8, New York, NY, USA, 2013. ACM.

[2] M. Coudron. Mptcp netlink. https://github.com/teto/mptcpnetlink, Feb 2014.

[3] G. Detal and S. Barré. Flexible path managers for MPTCP. http://www.tessares.net/path-manager/.

[4] E. Dumazet and Y. Cheng. TSO, fair queuing, pacing: three's a charm. Presented at IETF'88, Nov. 2013.

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

[6] A. Ford, et al. Architectural Guidelines for Multipath TCP Development. RFC 6182, March 2011.

[7] A. Ford, et al. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824, January 2013.

[8] N. Handigol, et al. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 253–264, New York, NY, USA, 2012. ACM.

[9] S. Hätönen, et al. An experimental study of home gateway characteristics. In *IMC*, pages 260–7, New York, New York, USA, 2010. ACM Press.

[10] M. Honda, et al. Rekindling network protocol innovation with user-level stacks. *SIGCOMM Comput. Commun. Rev.*, 44(2):52–58, Apr. 2014.

[11] A. Huttunen, et al. UDP Encapsulation of IPsec ESP Packets. RFC 3948 (Proposed Standard), Jan. 2005.

[12] Y.-s. Lim, et al. 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] A. Müller, et al. Analysis and topology-based traversal of cascaded large scale NATs. In *HotMiddlebox*, pages 43–48. ACM Press, 2013.

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

[15] C. Paasch, et al. Exploring Mobile/WiFi Handover with Multipath TCP. In *ACM SIGCOMM CellNet workshop*, pages 31–36, 2012.

[16] C. Paasch, et al. Experimental evaluation of Multipath TCP schedulers. In *Proceedings of the 2014 ACM SIGCOMM Workshop on Capacity Sharing Workshop*, CSWS '14, pages 27–32, New York, NY, USA, 2014. ACM.

[17] Q. Peng, et al. Energy efficient Multipath TCP for mobile devices. In *Proceedings of the 15th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '14, pages 257–266, New York, NY, USA, 2014. ACM.

[18] C. Raiciu, et al. Improving Datacenter Performance and Robustness with Multipath TCP. In *ACM SIGCOMM 2011*, 2011.

[19] C. Raiciu, et al. 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.

[20] J. Salim, et al. Linux Netlink as an IP Services Protocol. RFC 3549 (Informational), July 2003.

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

[22] P. S. Schmidt, et al. 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.

[23] Y. sup Lim, et al. Cross-layer path management in multi-path transport protocol for mobile devices. In *INFOCOM, 2014 Proceedings IEEE*, pages 1815–1823, April 2014.

[24] H. Tazaki, R. Nakamura, and Y. Sekiya. Library operating system with mainline Linux kernel. Presented at Netdev 0.1, Feb 2015.