An Implementation of Multipath TCP in ns3

Matthieu Coudron, Stefano Secci

Sorbonne Universites, UPMC Univ Paris 06, UMR 7606, LIP6, Paris, France. Email: firstname.lastname@upmc.fr

Abstract

The Multipath Transport Control Protocol (MPTCP) is undergoing a rapid deployment after a recent and quick standardization. MPTCP allows a network node to use multiple network interfaces and IP paths concurrently, which can lead to several advantages for the user in terms of performance and reliability. In this paper, we describe an MPTCP implementation in the Network Simulator 3 (ns3), comparing it with both the Linux implementation and previous ns3 implementations. We show that it is compatible with the Linux implementation and that it has a desirable similar behavior in traffic handling. Our goal is to allow researchers develop and evaluate new features of MPTCP using our simulator in a much faster way than they would with a kernel implementation, hence boosting MPTCP research.

Keywords: MPTCP, Network Simulator, Implementation Evaluation
2016 MSC: 20-70, 20-80

1. Introduction

Nowadays modern mobile devices are usually equipped with several network interfaces: it may be WiFi and Ethernet for laptops, or WiFi and cellular for smartphones. In this context, a user may want to leverage these different interfaces into using concurrently several paths to achieve the following goals:

1. Seamless mobility: with legacy TCP, losing an IP address means losing active TCP sessions, which in a mobility scenario translates into a communication delay necessary to setup a new connection. With multipath transport, one device can establish several connections in advance and (re)transmit data on alternate paths when there is a partial or total failure on one path (see [1]). Bandwidth aggregation: The ability to aggregate the bandwidth of several links is also very appealing and appears as the most anticipated feature.
2. Higher confidentiality: if a flow of data is split over several paths, it may become harder for an attacker to reconstitute the whole connection flow.
3. Lower response time: sending duplicated packets on several paths can increase the probability for the data to follow uncongested paths.

More elaborate features can emerge from combining some of these techniques. For instance, a smartphone user may enable both LTE and WiFi interfaces to benefit from the mobility advantage, and at the same time to limit the cellular throughput to save some battery or money. Or one may choose to trade some of the aggregation benefit in exchange for higher confidentiality.

Yet a multipath protocol needs to address several problems to reach the previous goals and deliver better than singlepath performance. Multipath communications lead to an increased occurrence of out-of-order packet deliveries, which may generate worse performance than single path protocols [2], besides questioning the fair usage of the network. Information such as the Round Trip Time (RTT) or the packet sequence number are critical to mitigate these problems, and are already available at the transport layer. While the application layer could provide a similar or even better service, having a standard multipath transport protocol allows to spread such a knowledge and should ease multipath communications deployment.

MPTCP is such a multipath transport protocol that attempts to address these issues in a backward compatible way. As any new Internet protocol, MPTCP has to face an ossified Internet whose many middleboxes are typically configured to block any unknown protocol extension or any new protocol. MPTCP must also address the fairness issue, i.e., it should not get too much more bandwidth compared to legacy users, otherwise the protocol could be blocked by Internet providers. At the same time MPTCP ambitions to be as least as good as TCP in terms of throughput, which can prove challenging in some environments.

In the following, in Section 2 we describe MPTCP, presenting its main components, and describing its state machine. In Section 3 we motivate our effort, detailing our implementation characteristics, and describing our design methodology and comparing it with existing implementations. Section 4 reports an experimental evaluation of the simulator. We open source the code of the simulator in [18].

2. Multipath TCP

MPTCP is a TCP extension formalized in RFC 6824 [3]; the MPTCP working group at the Internet Engineering Task Force (IETF) was formed in october 2009; since the beginning, it emphasizes backwards compatibility with the network and the applications. This is an aspect to keep in mind when looking at some design decisions that may appear counter-intuitive at first (for instance the creation of an additional sequence number space or the requirement to wait for two levels of acknowledgements before being authorized to free the buffers). As a result, TCP applications can run unmodified with MPTCP. This differs from the Stream Control Transmission Protocol (SCTP) [4], a previous IETF effort, that provides more features but whose deployment is impeded by the

many middleboxes on the Internet, blocking unknown protocols. [1]

MPTCP should be pareto optimal, i.e., it should not harm any TCP user while improving the situation for MPTCP users. Achieving pareto optimality is still a problem for MPTCP [2] though improvements have been made [6]. Several techniques exist in the literature, such as watching the loss correlation between subflows to infer if they shared a bottleneck, but such methods make assumptions about the network that prevent them from being holistic. The conservative approach is to consider that all subflows share the same bottleneck: this is the so-called resource pooling principle [7]. Fairness violation and out-of-order packet delivery are two problems that any multipath protocol shall to solve.

2.1. High level design of MPTCP

MPTCP consists in a shim layer, as represented in Figure 1. It is built between the application and the TCP stack that unifies several TCP connections, called "subflows" in the MPTCP context. A subflow is a TCP connection characterized by a tuple (IP_{source}, TCP port_{source}, $\text{IP}_{destination}$, TCP $\text{port}_{destination}$) and is assigned a unique subflow id generated by the MPTCP stack. MPTCP uses such a subflow identifier to convey subflow related advertisements; it does not use the IP addresses as identifiers because they can be rewritten by external middleboxes. One can alternatively define an MPTCP connection as a set of one or many subflows aggregated to feature at least the same set of services as a singlepath TCP communication.

MPTCP signals information with its peer through the use of TCP options. To reorder traffic striped on several subflows, MPTCP adds a global Data Sequence Number (DSN) namespace shared among subflows and exchanged through TCP options. The DSN are then mapped to relative Subflow Sequence Number (SSN), i.e., the TCP subflow sequence numbers, through the Data Sequence Signal (DSS) (Data Sequence Signaling), and are acknowledged with what we refer to as **Data Ack** in the rest of this paper, exchanged through the same DSS option.

The RFC 6182 [8] lists a few functional goals that are deemed mandatory for a wide deployment of the protocol:

1. MPTCP must support the concurrent use of multiple paths. The resulting throughput should be no worse than the throughput of a single TCP connection over the best among these paths.
2. MPTCP must allow to (re)send unacknowledged segments on any path to provide resiliency in case of failure. It is advised to support "break-before-make" scenarii, e.g., buffer the data when a (mobile) user loses temporarily

[1] SCTP is now deployed mainly thanks to the WebRTC protocol but is tunneled over UDP packets [5]. SCTP proposed to opt-out some TCP services on a per connection basis such as in-order delivery. Ordering is indeed unnecessary when downloading an archive, because head-of-line blocking may slow down the connection.

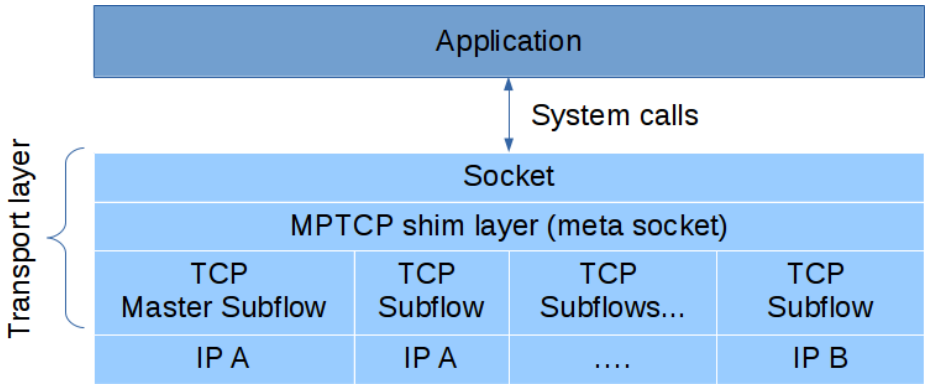


Figure 1: MPTCP: a shim layer in the stack. Subflows can share the IP address (using a different port) or have different IPs.

all connectivity, to allow resuming the communication as soon as a new subflow gets available.

[8] also lists three compatibility goals:

- The applications must be able to work with MPTCP without being changed, for instance via an operating system upgrade. It also implies that MPTCP keeps the in-order, reliable, and byte-oriented delivery. [2]
- MPTCP should work with the Internet as it is composed today, that is with middleboxes blocking unusual payloads or even modifying the payload such as internet accelerators, Network Address Translator (NAT) etc. The best way to achieve this is to appear as a singlepath TCP flow to the middleboxes. Hence MPTCP relies on TCP options for signaling. TCP option space is scarce (40 bytes maximum per packet).
- MPTCP should be fair to single path TCP flows at shared bottlenecks, i.e., not be greedier. At the same time, MPTCP still shall perform better.

As part of the network compatibility goal, MPTCP should provide an automatic way to negotiate its use, and upon failure of such a negotiation, fall back to legacy TCP. This fall back is also possible even after successful completion of the MPTCP handshake, in case no data ack is received during a certain time, or checksums are invalid.

2.2. Connection process

Initiation. Supposing that the MPTCP extension is not disabled, and that the application remained unchanged, the MPTCP connection is initiated through

[2] Nevertheless an extended API is being standardized in [9] for applications to squeeze more out of MPTCP.

the TCP socket interface via the **connect** system call. As per the MPTCP Linux system nomenclature, we call this first TCP connection the master connection. This call must generate a random key to be used during the TCP handshake as can be seen in Figure 2. This key is later hashed and used by MPTCP to authenticate additional subflows.

Once other subflows are established, the master subflow can be removed as any other and holds no specificity. Upon SYN reception, the server generates also a key which is reflected by the client in the final TCP handshake ack. This allows the server to operate in a stateless mode. Indeed an MPTCP stack needs to allocate more data structures than a legacy TCP connection to save the key, the list of subflows, their identifiers etc. For the sake of efficiency, the allocation of these data structures can be deferred until the moment the MPTCP negotiation succeeds.

Addition of other subflows. The host can open a new subflow as soon as a DSS option with a data ack is received, which requires at least two RTTs since the very first handshake. Hence the choice of the initial subflow can have an impact on the throughput, especially for short connections. Both the client and the server can create new subflows. Either the host initiates the new connection, or it advertises a couple (IP, port) that the peer can choose to connect to. The policies are local; for instance, in the Linux implementation, the server advertises its ports, but it lets the subflow creation initiative to the client because of NATs that could invalidate the client-advertised addresses. It is worth noting that several subflows can be created from the same IP address with different ports. This may prove worthwhile to exploit the network path diversity, in case the network runs load-balancing [10]. There is no standard procedure and the subflow opening/closing strategy depends on local policies. It may be wiser to let clients initiating the connection though, due to the presence of NATs. Subflow control can also be delegated to a third party controller [10] [11].

2.3. Congestion control

TCP fairness can be a controversial topic: a malicious TCP user who wants more bandwidth can create additional TCP connections (as many download accelerators do) to increase its share at the bottleneck. In the following, we consider well-behaved hosts since this is the usual framework priori to any congestion control reasoning.

Without specific congestion control algorithm, a multipath transport protocol would adopt a similar behavior at the bottleneck since being an end-to-end technology, it has no information over the topology. TCP users would see their bandwidth decrease and MPTCP deployement hindered. Under these conditions, how to achieve both fairness and higher throughput? Knowing if two subflows share a same resource (e.g., a link or a router) would allow to run a congestion control on sets of subflows. Clustering techniques, e.g., [12] and [13], have been developed to detect bottlenecks based on delay and loss patterns. Such techniques need to be foolproof as false negatives generate bandwidth stealing. This is a difficult task without the help from the network, as the heuristics

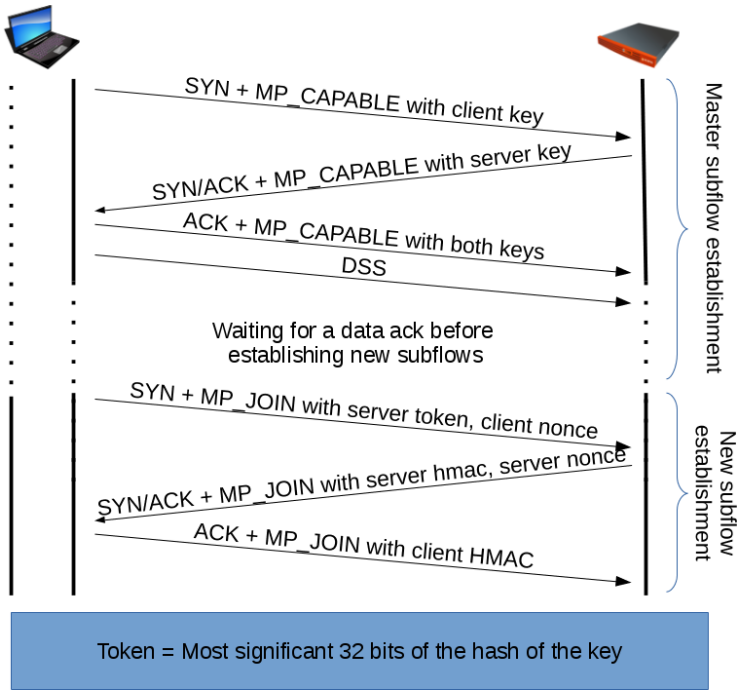


Figure 2: Illustration of used notations for two subflows.

need to work across a wide range of configurations, such as the router buffering policies. Their efficiency is also difficult to evaluate for the same reasons but even if a perfect scheme existed, relying on it depends on the fairness notion.

MPTCP embraces the resource pooling principle, which makes a collection of resources behave like a single pooled resource. This conservative approach considers that all subflows share a bottleneck and that their additive component should be coupled.

MPTCP congestion control algorithms modify the congestion avoidance phase of the TCP congestion control only: the decrease phase remains the same as in TCP. Several congestion control algorithms have been proposed such as Linked-Increase Algorithm (LIA [14]) or Opportunistic LIA [6] (OLIA). They couple the increase MPTCP congestion window with the congestion window of its subflows:

- $w_i = w_i + \min(\frac{a}{w_i}, \frac{1}{w_r})$ per acknowledgement on path i.
- $w_i = \frac{w_i}{2}$ per congestion event on path i.

with a being an aggressiveness factor updated once in a while (per window a priori) and equal to:

$$a = \frac{\max_r(\frac{w_i}{rtt_i^2})}{\sum \frac{w_i}{rtt_i}^2} * \sum_i w_i$$

with :

$$\begin{cases} w_i \text{ the window size on path i} \\ rtt_i \text{ the round trip time on path i} \end{cases} \quad (1)$$

The *min* in the first equation ensures that MPTCP is never more aggressive than TCP on a single path. It is important to remember that the advertised receive window is shared between subflows. As such, there may be cases where a subflow is capable of sending data, i.e., it has a free transmission window but there is no more space in the receive window - phenomenon known as Head-of-Line (HoL) blocking. This may happen when a feature called opportunistic retransmission is implemented [15], which in such cases retransmits data hoping to solve the HoL issue. Opportunistic retransmission can be used in conjunction with slow subflow penalization: if a subflow holds up the advancement of the window, MPTCP can reduce forcefully its congestion window along with its slow start threshold.

2.4. Scheduling

. The scheduler chooses when and on which subflow to send which packets. A good scheduler should attempt to reduce the probability of HoL blocking. For instance, opportunistic retransmission and penalization are reactive mechanisms that waste bandwidth. The Linux implementation currently includes two schedulers:

- The 'default' scheduler sorts subflows according to their RTT and sends packets on the first subflow with free window.
- A round robin scheduler that forwards packets in a cyclic manner on the first subflow with free window available.

Retransmission timeouts (RTO and delayed acks) need to be chosen with great care since a subflow RTO or out of order arrivals can provoke HoL blocking faster than in the single path case, as also explained in [16]. For instance, some of the state of the art schedulers propose to send packets out of order so that they can arrive in order [17].

2.5. MPTCP state machine

As a preliminary step before implementing MPTCP in ns3, we needed to formalize the current status of the standard to have a precise specification.

In particular, we had to extend the connection closure Finite State Machine (FSM) described in [3] to cover the whole protocol, i.e., while the active and passive close are presented as a diagram in [3], we extended the visual description to our interpretation of the standard. The result is depicted in Figure 3, which represents what appears to be the single full representation of the finite state machine of MPTCP.

While being similar to TCP, we chose to split the **ESTABLISHED** state into the **M_ESTA_WAIT** and **M_ESTA_MP** states to distinguish between a state where MPTCP waits for a first Data acknowledgement (DACK) and

a state where MPTCP can create additional subflows. We also mapped for each MPTCP state the states in which TCP subflows can be, as well as which MPTCP options could possibly be sent. The tabulated study report is available online [18].

2.6. Associated challenges

We already mentioned a few challenges in the previous sections. Our stance is that MPTCP is already robust enough by design to fulfill the network and application compatibility goals (as confirmed by the commercialization of successful MPTCP-based products developed by several large corporations such as OVH, Apple, Citrix).

About the requirements described in Section 2.1, the current specification and implementations adequately meet the resiliency requirement; when one link fails, retransmission of the packets is straightforwardly done on another subflow as per the basic scheduler behavior. The main obstacle to MPTCP deployment today remains the throughput and fairness goals. While there are examples of increased throughput through the use of MPTCP (e.g., the fastest TCP connection was done with MPTCP [19]), this requires specific conditions such as enough buffer and homogeneous paths; there are also cases, as in [2], where MPTCP performs worse than TCP on the best available path. This does not comply with the objective of doing always better than TCP. MPTCP must acquire the intelligence to distinguish when and which subflows to use to perform well. Reaching this goal is made even harder with the throughput goal since MPTCP is less aggressive than TCP on every subflow.

Path management is also a problem - though less studied - since creating many subflows with the hope of exploiting path diversity can hurt the performance (due to competition between subflows [10]). The problem is two-fold:

1. transport protocols being end-to-end, hosts do not know the topology;
2. even if the hosts knew the topology, they cannot enforce a forwarding path. Segmented routing may provide a partial solution in this regard.

As for wide area networks topologies, there usually is more than one path between source and destination. It can be because of intra-domain redundancy or because several ISPs compete on the same path. In this direction, there is ongoing work to exchange topology information between nodes that could solve point 1) above, for instance Path Computation Elements or at the ALTO (Application Layer Traffic Optimization) working group [20].

Topology is a critical information that operators may not be fond of leaking, hence some approaches look at how to provide an overview of the topology through sparsification techniques [21]. From the previous technologies, a host can deduce an optimal number of subflows, but this may prove pointless if the forwarding problem (point 2) above) is not solved. As such, solutions in locally controlled environments such as an SDN (Software Defined Network) datacenter seem appropriate.

Thus it is advised to use the correct number of subflows (MPTCP can create more subflows but mark them as backup subflows), no more no less, to reach the

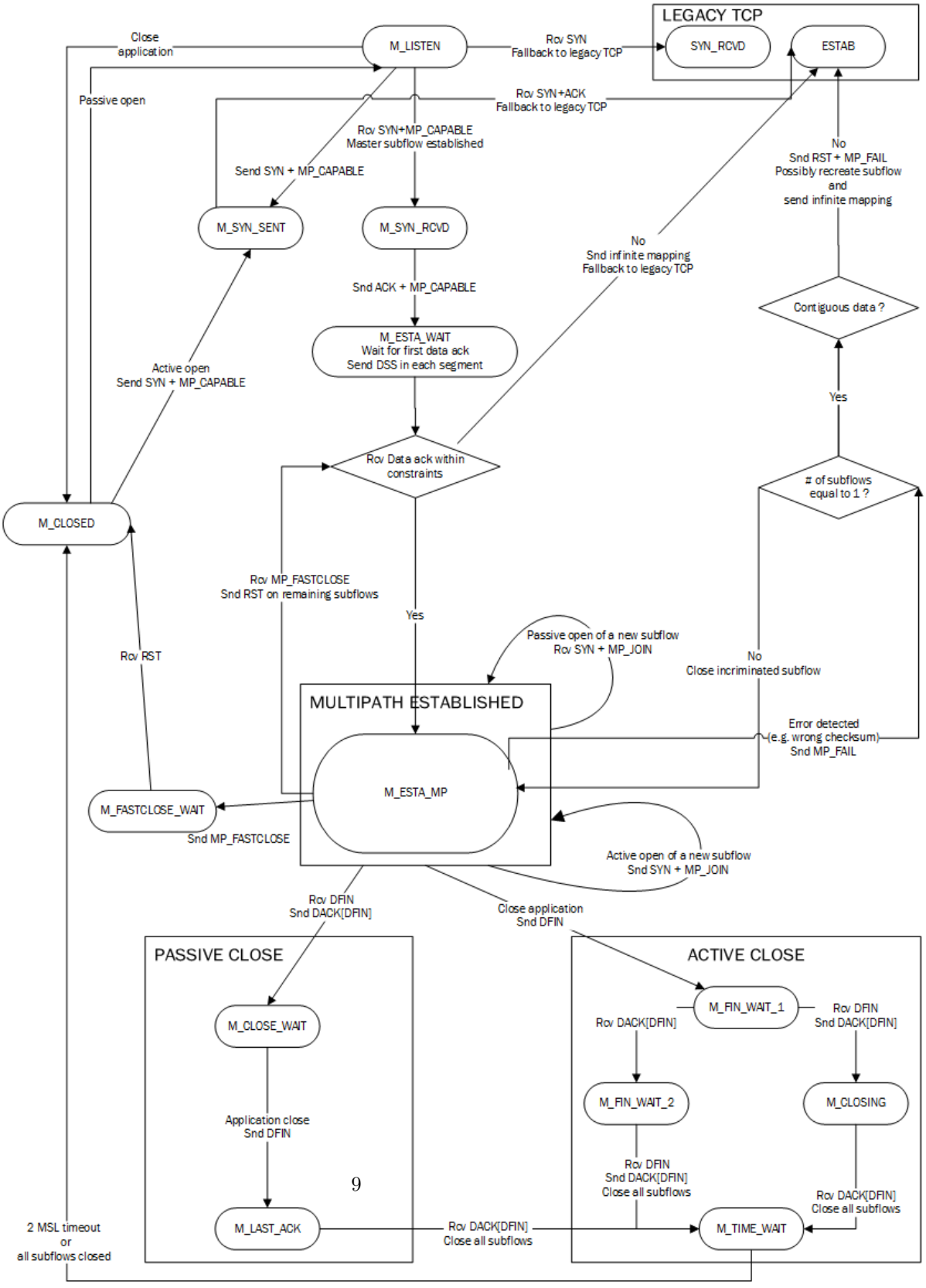


Figure 3: MPTCP state machine.

optimal throughput. The path management problem also explains why many of the commercial products embed MPTCP into proxy middleboxes (Gigapath[3], OVH[4], Tessares[5]); certainly they grant the benefits of MPTCP to legacy clients, but the middleboxes can also be better informed of the available path diversity thanks to their topological position.

Multipath transport incentives are not limited to throughput aggregation or reliability goals, and as such one could imagine modes where the cost of an interface can affect packet scheduling over interfaces as suggested in [22]. The cost could be given by the energy consumption of the interface or depending on its fare rate. The user could even set trade-off levels such as losing 30% of the optimal throughput if it allows for a fairer distribution between subflows. LEDBAT-multipath [23] is one of such alternative modes. Information that used to be of little interest with one path are now helpful in a multipath context. For instance, if the MPTCP layer is aware of the data emission profile, it can adapt the scheduling to favor throughput (bulk transfer) or schedule packets so that they arrive early at the receiver (at the end of a burst).

3. An MPTCP implementation in ns3

A few MPTCP implementations already exist, some of which already used in production environments such as Apple's voice recognition system Siri. Among the implementations, the Linux one[6] is the oldest one with some impressive achievements (Fastest TCP connection [19]) and likely used in all the commercial products presented in Section 2.6. Work is also done to improve the MPTCP support on other operating systems such as Solaris[7] and FreeBSD [8]. Hence asking why developing a MPTCP simulator is a legitimate question. In this section we describe our motivations and the technical aspects of our implementation. We also present a few tools we developed to ease testing and analysis of related MPTCP traces.

3.1. Presentation of ns3 and Direct Code Execution

Ns3 [24] is a popular network simulator in the networking research community as is confirmed by the two previous implementations. Its success is likely due to its General Public License and also because the technical base as well and the support team are trustworthy. It is best described as a C++ discrete time simulator, i.e., events are scheduled in the simulator time and once all events at the specific time are processed, the simulator updates the current time with the time of the next scheduled events. It allows the simulator clock to be independent from the wall clock, most of the times faster.

[3] https://www.ietf.org/proceedings/91/slides/slides-91-mptcp-5.pdf
[4] https://www.ovhtelecom.fr/overthebox/
[5] http://www.tessares.net
[6] `http://multipath-tcp.org`
[7] `https://mailarchive.ietf.org/arch/msg/multipathtcp/ugMIu566McQMn8YCju-CTjW9beY`
[8] `http://caia.swin.edu.au/urp/newtcp/mptcp`

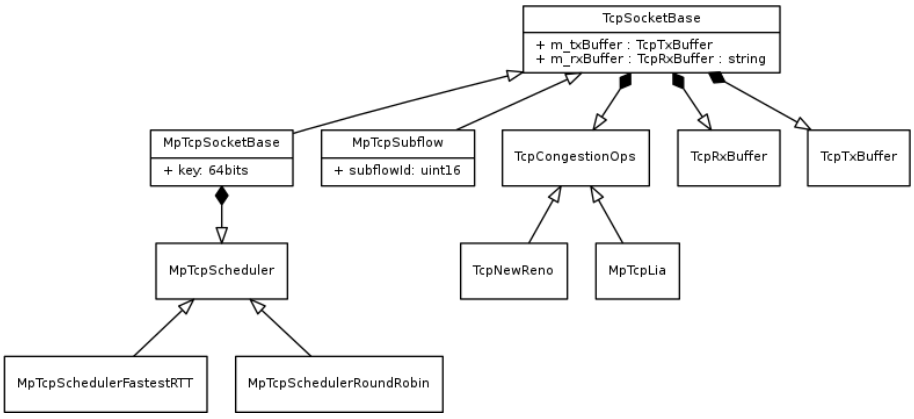


Figure 4: Implementation structure in ns3 code.

Direct-Code Execution (DCE) is an ns3 extension that allows to load applications compiled with specific options (as well as a fork of the Linux kernel [25]) within the ns3 environment. The advantage is that the simulation runs in discrete time and thus provides results independently of the host CPU. As a comparison, the fidelity of mininet[9], a container-based simulator, decreases inversely with the processing load [26].

3.2. Why an MPTCP simulator?

Simulation traditionally comes handy to (i) run experiments faster, and to (ii) focus the research efforts on the algorithmic part rather than implementation complexity.

Experimenting with MPTCP in the real world can be complex depending on the scenario. Mobility is a major use case and usually requires access to cellular (4G) and WiFi interfaces. Not only does it have a cost but 4G is not ubiquitous and experiments involving wireless channels are time consuming because of the variability and care their setup require. Other experiments may want to assess the behavior under realistic circumstances in terms of subflow latencies, and one way to do so is to rely on accurate path time and latency measurements (e.g., to measure one-way delays as in [27]). Exploiting such traces can prove very challenging in real setups, but are straightforward in discrete time simulators.

Besides the obvious huge time gain in both experiments design and execution time, focusing the research effort on the algorithmic details, e.g., the congestion control algorithm, the scheduler, the buffer dimensioning, is also an important factor when deciding whether using a simulator or a real operating system implementation, especially when looking back at the number of use cases described in Section 2.6. Implementing such solutions into current operating systems usually means adding the features into the kernel. While simulation results may

[9] `http://mininet.org/`

Features	Chihani et al. [28]	Kheirkhah et al. [29]	Our implement.
Option serialization	Partial	Partial	Full
Standard compliance	Connection phase	Connection phase	Full
Backward Compatibility	No	No	Yes
Ack-aware buffer mgnt	No	No	Yes
Comparison to OS implem.	No	No	Yes

Table 1: Comparison between ns3 MPTCP simulators.

lose fidelity compared to a reasonable kernel implementation, we argue that kernel development complexity can generate bad implementations that can not be easily verified and may not be representative of expected results/analytical models. In those cases, using a simulator model beforehand is reasonably faster. The usage of an MPTCP simulator can ease reproducibility and can help realize problems ahead of time.

Last but not least, we also think the implementation can serve for education purposes since the model only deals with MPTCP essentials, thus reducing the learning complexity.

3.3. Related work

We have been able to access two previous MPTCP implementations, [28] and [29], both done using ns3 as well. These two implementations are similar in many aspects and are compared with ours in Table 1.

Recent developments in ns3 such as TCP option support and generic packet serialization in a wire format made it possible for ns3 to communicate with real stacks. Contrary to previous ns3 implementations that support a subset of the options, ours support full (de)serialization of MPTCP options, which means it can handle a higher variety in options (e.g., 32 and 64 bits encoding for DSNs).

To allow the communication with an external stack such as the linux one, we also implemented standard compliant connection and closing phases, which is another differentiating point from [28] & [29]. Thus our implementation is capable of generating valid tokens based on the (sha1) hash of a random key, and closing a connection requires the sending and acknowledgement of a DSS with the data FIN bit. While the implementation is not robust enough yet to handle all cases, it managed to exchange a file with an external linux MPTCP stack with the use of DCE as reported hereafter.

Contrary to [28] & [29], our implementation is backward compatible with existing ns-3 TCP scripts, following the MPTCP standardization spirit. Thus in our implementation, the connection phase starts with a legacy TCP socket (more precisely a 'TcpSocketBase' see Figure 4) and only once an MPTCP option is received it evolves into an MPTCP socket (see 'MPTCPTcpSocketBase'

in Figure 4). This allows for better integration with the general framework, and adds the additional benefit of allowing the MPTCP connection to fall back to TCP. Our hope is to be able to upstream this implementation so that improvements can then be added incrementally.

We also respected an aspect of the specification that could affect the simulation fidelity, i.e., data can not be removed from the subflow sockets until it is acknowledged at both the TCP and MPTCP levels.

Finally, our implementation is also the first to our knowledge to be evaluated against an operating system stack in comparable conditions as described later in Section 4.

3.4. Supported and missing features

It is worth noting that Table 1 does compare the three implementations with respect to high-level aspects, without delving in a precise list of features. It is however worth mentioning the lack of support in [28, 29] of many key features needed to draw realistic settings, such as asymmetric routing, subflow-level buffer management, the possibility to select single-path TCP congestion control algorithms, and the existence of an interface for the scheduler. All these features are supported by our implementation.

The implementation was developed in ns-3.23 while giving care to performance and algorithmic aspects. As such, the fallback capabilities (MP_FAIL option, infinite mapping and checksums) of the protocol have not been implemented with the exception of the initial fallback, when the server does not answer with an MP_CAPABLE option, i.e., it does not support MPTCP and the client falls back to legacy TCP. This was made possible by extending the existing ns3 code infrastructure; for instance in Figure 4, only the structures starting with "MpTcp" were added. It also spares some resources during the simulation. Indeed the ability to enable dynamically MPTCP on a per connection basis means that our implementation works with all the other TCP scripts. This obviously implies that we inherit the legacy behavior of TCP in ns3, including desirable features such as the possibility to configure asymmetric link and routing properties. Moreover, one can infer from Figure 4 that new schedulers can be easily interfaced to the simulator and that MPTCP-level buffers can be reconfigured too.

We focused our work on implementing the aspects that could have an impact on the performance such as how data is freed from the buffers: MPTCP requires the full mapping to be received before being able to free the buffer. We detail and describe a list of supported key features of our implementation in Table 2.

Compared to the linux implementation, a major shortcoming of the Network Simulator 3 (NS-3) MPTCP implementation is the lack of the penalization mechanism, which reduces the window of a subflow that blocks the MPTCP window and the opportunistic retransmission feature.

Also contrary to the linux implementation that generates DSS mappings just in time to be able to adapt to network conditions, we designed the scheduler to be able to delay the decision until the last minute or to create mappings

SHA1 support	We added an optional SHA1 support in ns3 to generate valid MPTCP tokens and initial DSNs. This allows to communicate with a real stack and also proved necessary for wireshark to be able to analyze the communication.
Scheduling	The fastest RTT and round robin schedulers are available.
Congestion control	Subflows can be configured to run TCP ones such as NewReno or LIA.
Mappings	As in the standard, data is kept in-buffer as long as the full mapping is received. This is necessary when checksums are used, otherwise this can be disabled to forward the data faster.
Subflow handling	It is done directly by the application that can choose to advertise/remove/initiate/close a subflow at anytime if it is permitted by the protocol.
Packet (de)serialization	Packets generated along with MPTCP options can be read/written to a wire, allowing an ns3 MPTCP stack to interact with other MPTCP stacks, such as a linux one.
Fallback	If the server does not answer with an MP_CAPABLE option, the client falls back to legacy TCP. Other failures are not handled, e.g., infinite mapping or MP_FAIL handling as simulating these features is of little interest.
Buffer space	Buffer space is not shared between subflows, data is replicated between the subflow and the meta send/receive buffers rather than moved.
Path management	We drifted away from the specifications in order to be able to identify a subflow specifically, i.e., we associate a subflow id to the combination of the IP and the TCP port. Nevertheless the implementation is modular so it is possible to replace the subflow id allocation with a standard scheme.

Table 2: List of supported and missing features.

in advance. Creating mappings in advance has the advantage of being able to generate mappings that cover several packets. While the throughput gain is negligible, it can spare some of the scarce TCP option space.

4. Evaluation

We present a simple use case where we compare the linux MPTCP implementation to our NS-3 stack. We chose not to run quantitative tests with the previous NS-3 implementations since they are based on NS-3 versions that date back from late 2009 for [28] (ns-3.6) to late 2013 for [29] (ns-3.19). This gap in versions make the practical evaluation a challenge as well as the interpretation of results, because the ns-3 TCP implementation significantly evolved in the meantime. Hence we tried to choose tools that would allow for seamless testing and analysis between the kernel and ns3 stacks to lighten the burden analysis. We had to do some more development to unify the linux and ns3 evaluation, leveraging on the standardized "pcap "format.

4.1. Used tools

As far as MPTCP signaling and data analysis is concerned, there is currently little choice, with only one tool we are aware of: MPTCPTRACE [30]. MPTCPTRACE is interesting for bulk analysis but we wanted to be able to look at the packet level to ease debugging. Thus we chose to improve the MPTCP support of wireshark [18], which specializes in packet-level network protocol analysis. A capture is on Figure 5. We mainly added the following features:

- MPTCP connection identification: ability to map TCP subflows together based on the key and tokens respectively sent in the MP_CAPABLE and MP_JOIN options.
- Verification of the initial DSN based on the MPTCP key.
- Display relative DSN, i.e., the first MPTCP sequence number sent being considered as 0.
- Computation of the latency between the arrival of new data throughout all subflows.
- Detection of DSS mappings spanning several packets.
- Detected retransmissions across subflows.

We wrote a tool called MPTCPANALYZER [31, 18] that leverages these results to produce the plots presented in the next section.

We present in the following a few simulations to compare the linux kernel implementation to our NS-3 implementation. In order to minimize the differences due to the environment and for the ease of reproducibility, we chose to compare the linux and ns3 MPTCP implementations within the DCE 1.7 framework. This means that nodes, routers and links are created by ns3. Every node can be configured with a specific network stack. We always install linux stacks in the routers.

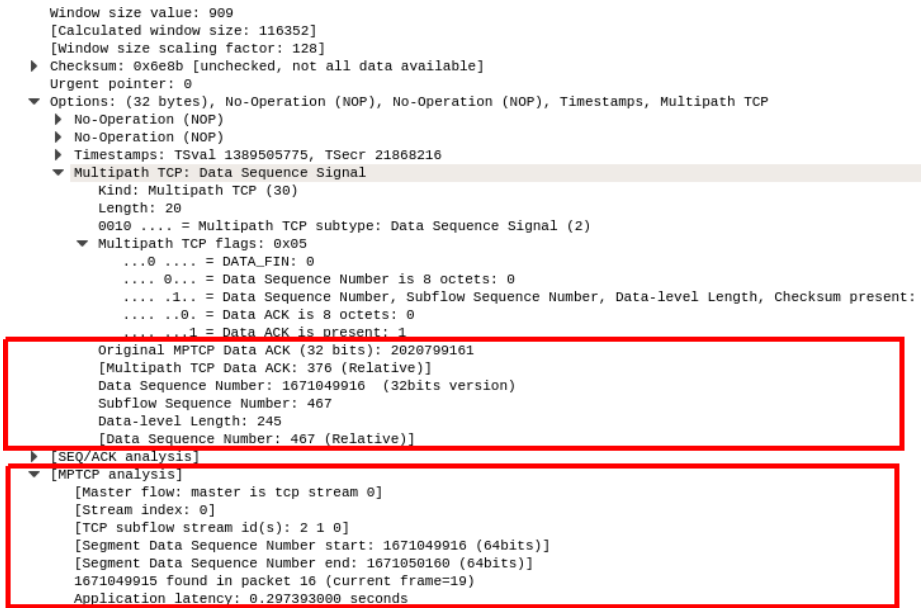


Figure 5: The wireshark MPTCP analysis section. Framed in red some of our additions.

4.2. Comparison with linux MPTCP implementation on a 2-link topology

The BDP refers to the number of unacknowledged bytes that can be inflight. It is generally advised to set the BDP higher than $RTT * \text{bottleneck capacity}$ to account for queuing delays in both the networks and the hosts. Note that in this case, as DCE runs in discrete time, kernel operations are virtually instantaneous if not programmed otherwise, so only the network latency impacts the RTT. On one path with a bottleneck of 2 Mbps and a RTT of 60 ms, the BDP is about 120 kbits. We run the experiments with LIBOS [25] applied against the linux MPTCP kernel v0.89.

Moreover:

- The scheduler is set to the round robin one.
- The number of paths is set to one (Figure 7a), then two (Figure 7b).
- The forward and backward one-way delays are set to 30 ms on each path.
- We execute using different receiver windows.

We ran 5-second IPERF2 [10] sessions between the two hosts without any background traffic on the topology of Figure 6. The size of the router buffers is the default linux one.

[10] `http://iperf.sourceforge.net`

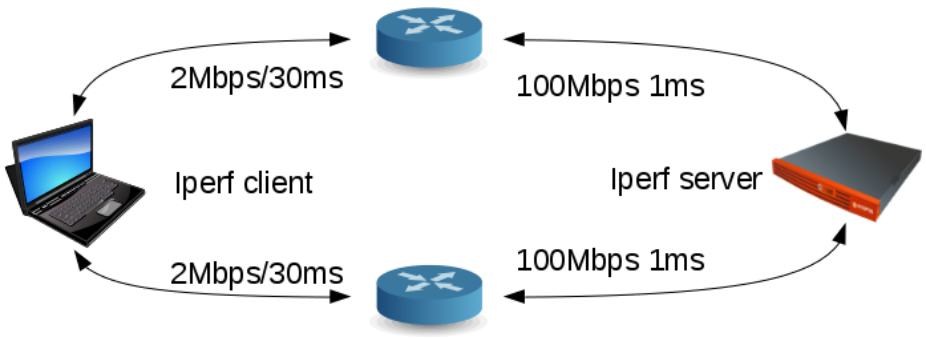


Figure 6: The topology used for the simulations.

In Figure 7a, we notice that both stacks make the maximum use of the paths except when it is window limited as for the 10 KB case. We can also notice that the throughput is a little more than the maximum throughput, which is likely due to IPERF2. Compared to the one path case, in the two paths case in Figure 7b we get the expected doubling in throughput when the window is big enough. It also seems that the ns3 version is greedier, namely in the 30 KB window case.

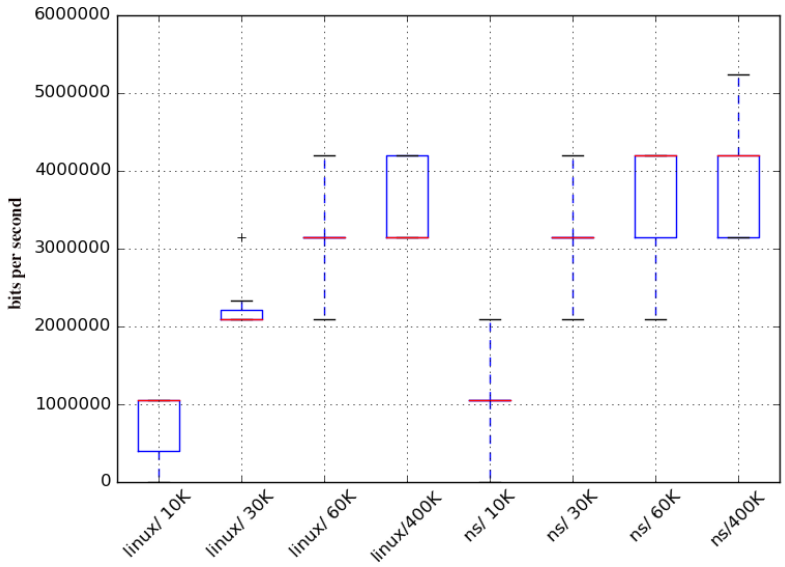
In order to check the behavior of the scheduler and thanks to MPTCPANALYZER, we were able to plot the relative MPTCP sequence numbers transmitted on every subflow for a 40 KB setup. We establish that DSNs are indeed sent in a round robin manner in both both the linux (Figure 8b) and the ns3 cases (Figure 8a). There are more sequence number for the ns3 case because the throughput was higher for that setup.

4.3. Open Problems

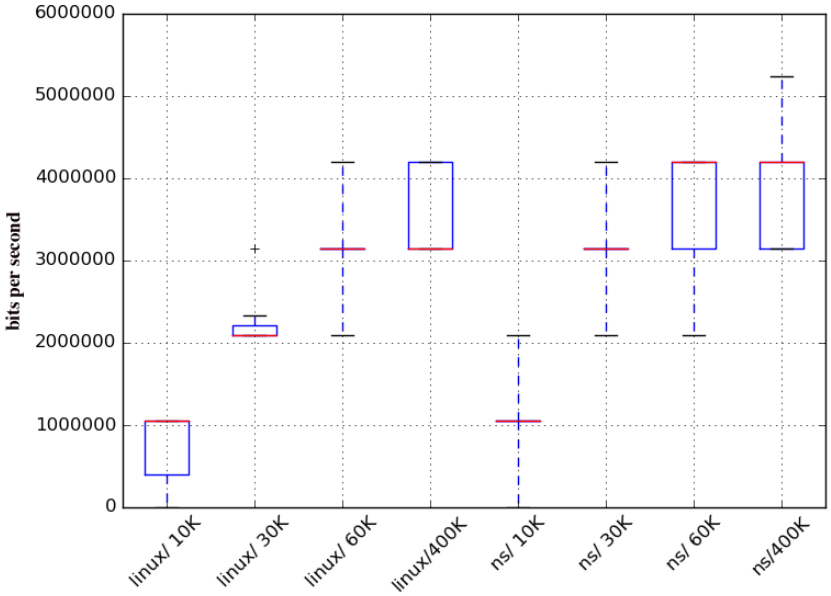
Limitations of the current simulations. The current buffer handling in ns3 currently copies data back and forth between the subflows and the meta socket instead of sharing a pool of memory. This is the main difference with other implementations, which could impact the simulation fidelity in tight buffer simulations. One promising solution is Non-Renegotiable Selective Acknowledgements (NR-SACK) [32]; but sadly the source is not available and this would require ns3 to implement SACK first.

Future work. Authors of [33] made an important contribution in applying experimental design to test the Linux stack over a large combination of configurations (buffer size, delay, loss, etc): we hope the experiment could be ported to work with DCE, which would remove the CPU bias for high loads.

Moreover, another interesting usage of our simulator may be on network coding usage within MPTCP. Network coding is an active area of research, which could improve MPTCP characteristics [34]. While operating system seem

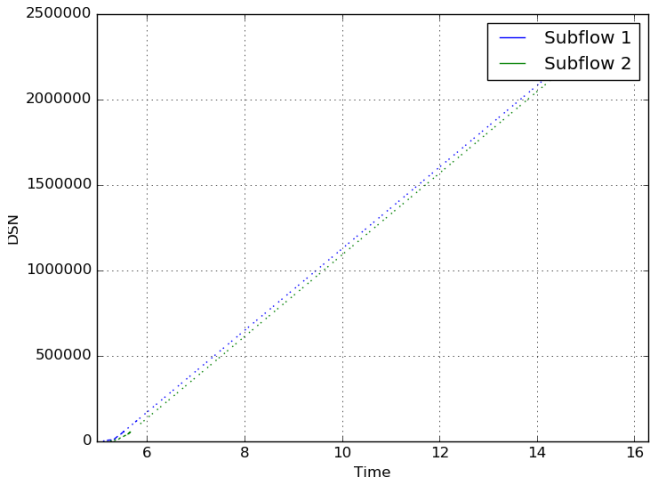


(a) Single path.

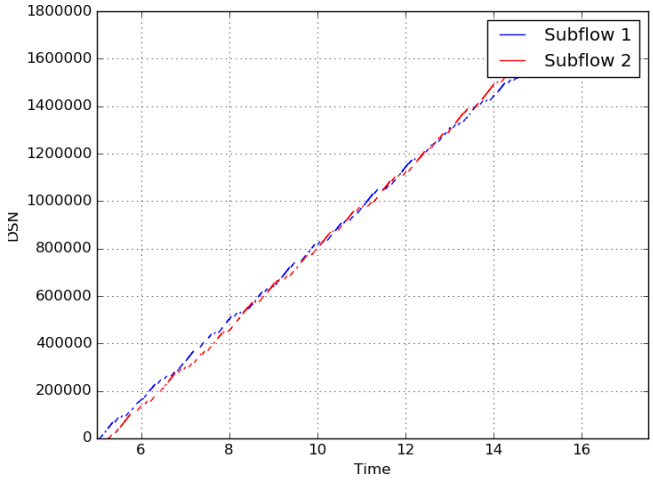


(b) Two paths.

Figure 7: MPTCP linux kernel and ns3 throughput comparison using IPERF2. Each boxplot indicates the min, 1st quartile, median, 3rd quartile and maximum. The x-label indicates the system used and the window size in KB.



(a) ns3.



(b) linux kernel.

Figure 8: Repartition of sequence numbers across two subflows with the round robin scheduler.

to remain oblivious to network coding, there exists a detailed library for ns3 [11].

5. Conclusion

We presented the MPTCP protocol and its new implementation we developed in the network simulator ns3. We described the MPTCP state machine we implemented and how our implementation conforms to many of the features described by the standard. We qualitatively compared our implementation to previous ns3 available implementations. We quantitatively compared it to the linux kernel implementation.

We hope our effort will allow to develop and experiment new schemes and features in an easier way, in order to improve or find new ways of using a multipath transport communication. Indeed MPTCP represents a subset of how multipath protocols could improve our future communications; it may represent a turning point between TCP and SCTP for instance. Relaxing some constraints such as the ordered delivery makes sense for bulk transfers and hopefully network programming interfaces will evolve to provide a smooth transition to multipath protocols.

We open source the code of the simulator in [18].

Acknowledgments

Thanks to M. Kheirkhah for open sourcing his MPTCP source code, Tom Henderson, Hajime Takizaki for the ns3 and DCE support. Thanks to Lynne Salameh for finding and fixing bugs. Thanks also to Olivier Bonaventure for answering our many questions on the MPTCP standard.

References

[1] A. Croitoru, D. Niculescu, and C. Raiciu, "Towards wifi mobility without fast handover", in *Proc of USENIX NSDI 2015*.

[2] S. Ferlin, T. Dreibholz, and O. Alay, "Multi-path transport over heterogeneous wireless networks: Does it really pay off?" in *Proc. of IEEE GLOBECOM 2014*.

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

[4] R. Stewart, Q. Xie, K. Morneault, and Al., "Stream Control Transmission Protocol", *RFC2960*, IETF, 2000.

[5] R. Jesup, S. Loreto, and M. Tuexen, "WebRTC Data Channels", draft-ietf-rtcweb-data-channel-13, IETF, 2015.

[11] http://kodo-ns3-examples.readthedocs.org

[6] R. Khalili, N. Gast, M. Popovic, and J.-Y. Le Boudec, "MPTCP is not pareto-optimal: Performance issues and a possible solution", *Networking, IEEE/ACM Transactions on*, vol. 21, no. 5, pp. 1651–1665, Oct 2013.

[7] D. Wischik, M. Handley, and M. B. Braun, "The resource pooling principle", *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 5, pp. 47–52, Sep. 2008.

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

[9] M. Scharf and A. Ford, "Multipath TCP Application Interface Considerations", RFC 6897, IETF, 2013.

[10] M. Coudron, S. Secci, G. Pujolle, P. Raad, and P. Gallard, "Cross-layer cooperation to boost multipath TCP performance in cloud networks", in *Proc. of IEEE CLOUDNET 2013*.

[11] R. van der Pol, S. Boele, F. Dijkstra, A. Barczyk, G. van Malenstein, J. Chen, and J. Mambretti, "Multipathing with MPTCP and openflow", in *Proc. of High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion*.

[12] S. Hassayoun, J. Iyengar, and D. Ros, "Dynamic window coupling for multipath congestion control", in *Proc. of IEEE ICNP 2011*.

[13] S. Ferlin, A. Alay, D. A. Hayes, T. Dreibholz, and M. Welzl, "Revisiting Congestion Control for Multipath TCP with Shared Bottleneck Detection", in *Proc of IEEE INFOCOM 2016*.

[14] H. Oda, H. Hisamatsu, and H. Noborio, "Design and evaluation of hybrid congestion control mechanism for video streaming", in *Proc. of IEEE CIT 2011*.

[15] C. Paasch, "Improving Multipath TCP", Ph.D. dissertation, Univ. catholique de Louvain, Belgium, 2014.

[16] M. Li, A. Lukyanenko, S. Tarkoma, and A. Yla-Jaaski, "The delayed ack evolution in MPTCP", in *Proc. of IEEE GLOBECOM 2013*.

[17] F. Mirani and N. Boukhatem, "Evaluation of forward prediction scheduling in heterogeneous access networks", in *Wireless Communications and Networking Conference (WCNC), 2012 IEEE*, April 2012, pp. 1811–1816.

[18] LIP6-MPTCP open source project repository (including the MPTCP ns3 simulator). [Online]. Available: https://github.com/lip6-mptcp

[19] C. Paasch, G. Detal, S. Barré, F. Duchêne, and O. Bonaventure, "The fastest TCP connection with Multipath TCP", 2013. [Online]. Available: http://multipath-tcp.org/pmwiki.php?n=Main.50Gbps

[20] Application Layer Traffic Optimization (ALTO) IETF working group. [Online]. Available: http://datatracker.ietf.org/wg/alto/charter/

[21] M. Scharf, G. Wilfong, and L. Zhang, "Sparsifying network topologies for application guidance", in *Proc. of IFIP/IEEE IM 2015*.

[22] S. Secci, G. Pujolle, T. M. T. Nguyen, and S. C. Nguyen, "Performance cost trade-off strategic evaluation of multipath tcp communications", *Network and Service Management, IEEE Transactions on*, vol. 11, no. 2, pp. 250–263, June 2014.

[23] H. Adhari, S. Werner, T. Dreibholz, and E. Rathgeb, "Ledbat-mp – on the application of lower-than-best-effort for concurrent multipath transfer", in *Proc. of WAINA 2014*.

[24] "Ns3 official website." [Online]. Available: www.nsnam.org

[25] T. Hajime, R. Nakamura, and Y. Sekiya, "Library Operating System with Mainline Linux Network Stack", in *netdev0.1*, 2015.

[26] H. Tazaki, F. Urbani, E. Mancini, M. Lacage, D. Camara, T. Turletti, and W. Dabbous, "Direct Code Execution: Revisiting Library OS Architecture for Reproducible Network Experiments", in *Proc. of ACM CoNEXT 2013*.

[27] M. Coudron, S. Secci, and G. Pujolle, "Differentiated pacing on multiple paths to improve one-way delay estimations", in *Proc. of IFIP/IEEE IM 2015*.

[28] B. Chihani and D. Collange, "A multipath TCP model for ns-3 simulator", *CoRR*, vol. abs/1112.1932, 2011. [Online]. Available: http://arxiv.org/abs/1112.1932

[29] M. Kheirkhah, "Multipath tcp in ns-3", Apr. 2015. [Online]. Available: http://dx.doi.org/10.5281/zenodo.32691

[30] B. Hesmans and O. Bonaventure, "Tracing multipath TCP connections", in *Proc. of ACM SIGCOMM 2014*.

[31] M. Coudron, "mptcpanalyzer: a multipath TCP analysis tool", Jun. 2016. [Online]. Available: http://dx.doi.org/10.5281/zenodo.55288

[32] F. Yang and P. Amer, "Non-renegable selective acknowledgments (nr-sacks) for mptcp", in *Proc. of WAINA 2013*, March 2013.

[33] C. Paasch, R. Khalili, and O. Bonaventure, "On the benefits of applying experimental design to improve multipath TCP", in *Proc. of ACM CoNEXT 2013*.

[34] M. Li, A. Lukyanenko, and Y. Cui, "Network coding based multipath tcp", in *Proc. of IEEE INFOCOM Workshops 2012*.