HAL
open science

Multipath TCP in ns3: Implementation Evaluation

Matthieu Coudron, Stefano Secci

To cite this version:

Matthieu Coudron, Stefano Secci. Multipath TCP in ns3: Implementation Evaluation. 2016. hal-01382907v1

HAL Id: hal-01382907

https://hal.sorbonne-universite.fr/hal-01382907v1

Preprint submitted on 17 Oct 2016 (v1), last revised 8 Jan 2017 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Multipath TCP in ns3: Implementation Evaluation

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 Performance

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 paths (see [1]).

2. Bandwidth aggregation: The ability to aggregate the bandwidth of several links is also very appealing and appears as the most anticipated feature.
3. 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.
4. Better average response time: sending duplicated packets on several paths increases the probability for the data to follow uncongested paths. Hence,

More elaborate features can emerge from combining some of these techniques. For instance, a smartphone user may enable both LTE and WiFi to benefit from the mobility advantage and at the same time limit the cellular throughput to save some battery or because WiFi is actually cheaper. Some other user 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 more out-of-order packet deliveries, which may lead to worse performance than single path protocols [2], and question 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 mutualize the knowledge and should ease multipath communications deployment.

MPTCP is such a multipath transport protocol attempt to address these issues in a backward compatible way. As any new Internet protocol, MPTCP is confronted with 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 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 detail our motivation to implement MPTCP in ns3. In Section 3 we detail our implementation characteristics and qualify its performance in different scenarii.

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 applications. This is an aspect to keep in mind when looking at some design decisions that may seem as 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) (SCTP [4]) that provides more features but whose deployment is impeded by the many middleboxes on the Internet, blocking unknown protocols. [1]

MPTCP must be pareto optimal, i.e. it must 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:

[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, head-of-line blocking may slow the connection.

this is the so-called resource pooling principle [7]. Fairness and the out of order packet delivery are two problems that any multipath protocol have to solve.

2.1. High level design of MPTCP

MPTCP consists in a shim layer as it can be seen on 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 this subflow id to convey subflow related advertisements since IPs are not reliable: they may be rewritten by external middleboxes. We can alternatively define an MPTCP connection as a set of one or many subflows aggregated to feature at least the same set of service 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 RFC6182 [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 all connectivity, to allow resuming the communication as soon as a new subflow gets available.

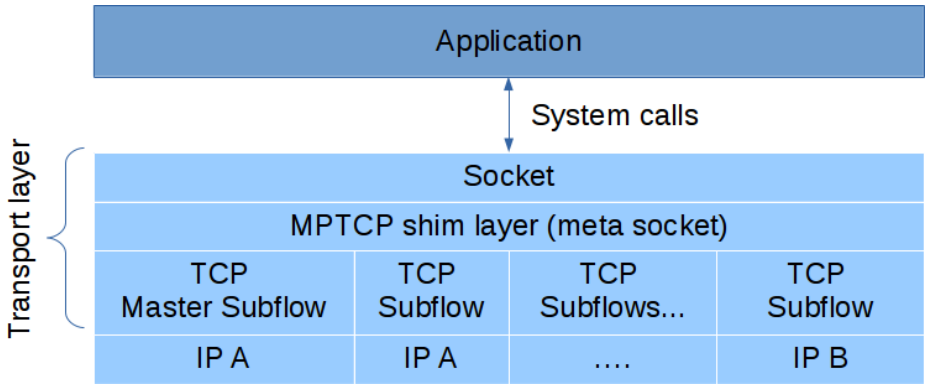


Figure 1: MPTCP: a shim layer in the stack. Subflows can share the IP address (using a different port) or have different IPs.

Rather than adding new features as SCTP does, MPTCP guidelines focus at not being worse than TCP and on wide deployment problems. [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 do 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 auto-

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

matic 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 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 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 ids etc. For 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, all the more important 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 and for instance in the Linux implementation, the server advertises its ports but let 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

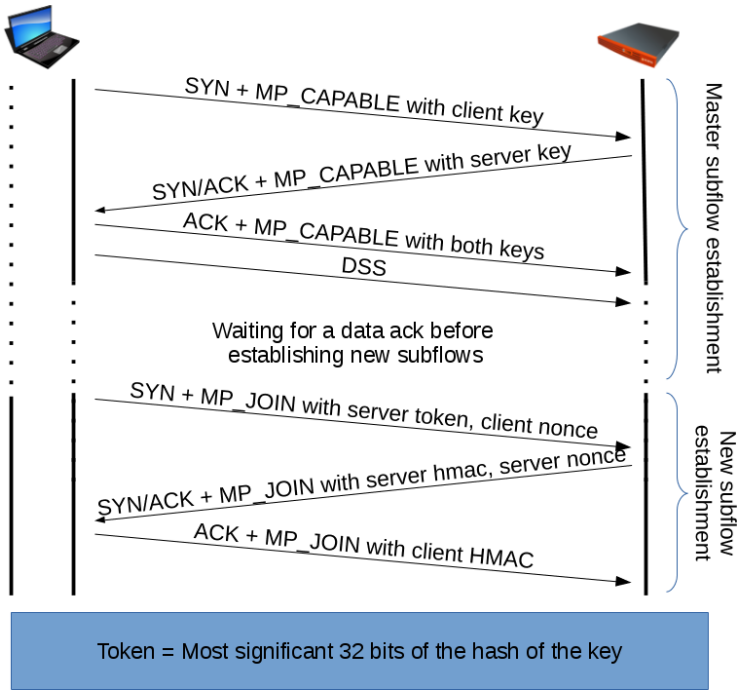


Figure 2: Illustration of used notations for two subflows.

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 initiate 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 additionnal 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 the fairness and higher throughput ? Knowing if two subflows share/make use of a same resource (e.g., a link or a router) would allow to run a congestion control on sets of subflows. Clustering techniques ([12] and [13] for instance) have been developed to detect bottlenecks based on delay and loss patterns. These techniques need to be foolproof as false negatives generate bandwidth stealing. This is a difficult task without help from the network as the heuristics need to work across a wide range of configurations, such as the router buffering policies etc... 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 (also called network fairness) 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 controls modify the congestion avoidance phase of the TCP congestion control only: the decrease phase remains the same as in TCP. Several congestion control have been proposed such as Linked Increase Algorithm (LIA [14]) or Opportunistic LIA [6] (OLIA). 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 :

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

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. has free window, but there is no more space in the receive window. This may happen when a feature called opportunistic retransmission is implemented [15], which in such cases retransmits data hoping to solve the head of line blocking. 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 head of line blocking (HoL). For instance opportunistic retransmission and penalization are reactive mechanisms that waste bandwidth. The Linux implementation we used included 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, it is needed to formalize the current status of the specifications. We extended the connection

closure Finite State Machine (FSM) described in [3] to cover the whole protocol in Figure 3, 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. 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) can create additionnal subflows and 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 trust of several companies such as OVH, Apple, Citrix that developed MPTCP-based products).

The resiliency goal 2. mentioned in Section 2.1 has also been proven to work, i.e. when one link fails, retransmission of the packets is done on another subflow. The main obstacle to MPTCP and multipath deployment protocols 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 can not 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-redundancy or because several ISPs compete on the same path. 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) datacenters 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 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 middle boxes also know more about the network diversity.

If the throughput goal mentioned in Section 2.1, one can imagine a realm of other possibilities for multipath protocols. Multipath incentives do not stop to throughput aggregation and as such one could imagine modes where the cost of an interface helps choosing over which interface to send packets as in [22]. The cost could be given by the energy consumption of the interface or depending on its fare rate. The user could also set tradeoffs such as losing 30%

[3] https://www.ietf.org/proceedings/91/slides/slides-91-mptcp-5.pdf
[4] https://www.ovhtelecom.fr/overthebox/
[5] http://www.tessares.net

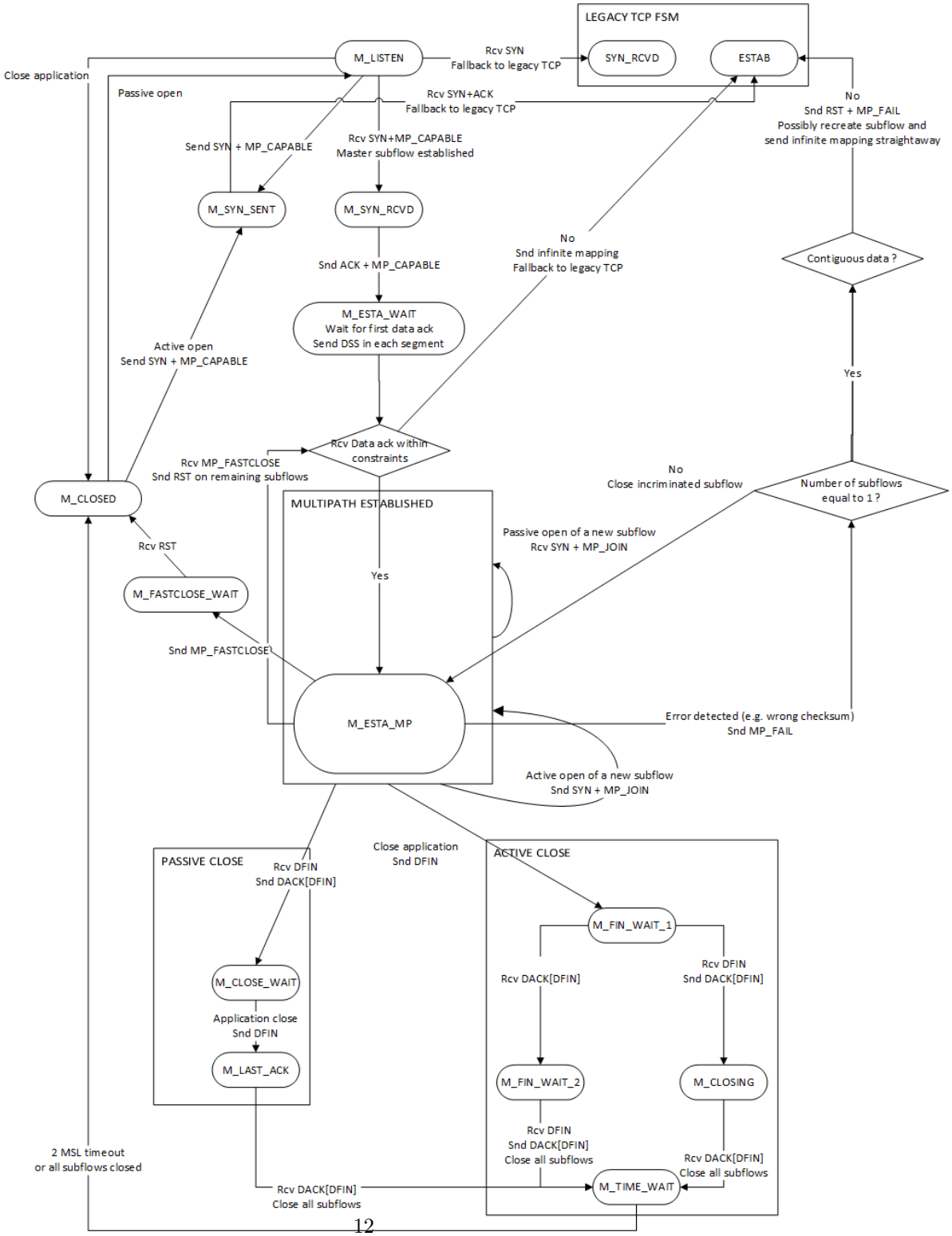


Figure 3: MPTCP state machine.

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 used in production environments such as Apple's voice recognition system Siri. Among the implementations, Linux[6] is the oldest one with some impressive achievements (Fastest TCP connection [19]) and is 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 the 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 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.

[6] http://multipath-tcp.org

[7] https://mailarchive.ietf.org/arch/msg/multipathtcp/ugMIu566McQMn8YCju-CTjW9beY

[8] http://caia.swin.edu.au/urp/newtcp/mptcp/

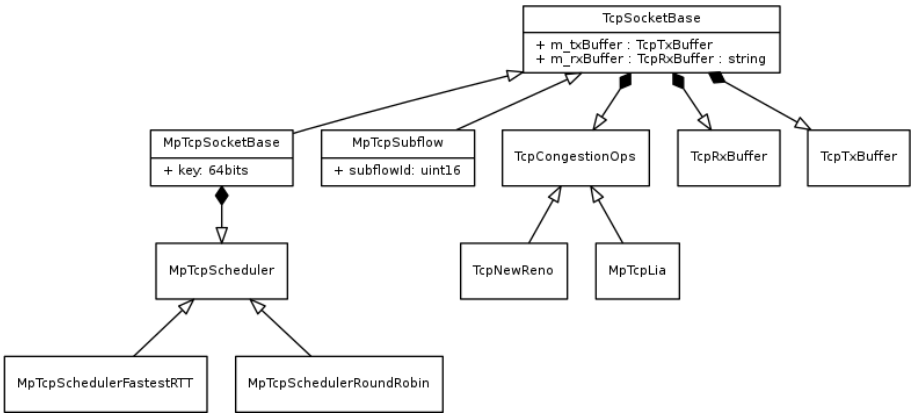


Figure 4: Implementation structure in ns3 code.

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, mininet's fidelity, a container based simulator, decreases inversely with the processing load [26].

3.2. Why a simulator ?

Simulation traditionally comes handy for two reasons:

1. Running experiments in a simulated testbed allows for faster reproducibility avoiding hardware costs.
2. Focusing on the algorithmic part rather than implementation complexity. Implementation details can have an impact on the overall fidelity of the model. Hence being able to compare with a simpler model beforehand can help find out if the difference in performance stems from implementation details or from the algorithm.

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 LTE and wifi. Not only does it have a cost but LTE is not ubiquitous and experiments

involving wireless channels are time consuming because of the variability and care their setup require. Other experiments rely on accurate time measurements (e.g. to measure one-way delays as in [27]), which can prove challenging in real setups but are straightforward in discrete time simulators.

Simulations can help find and solve a problem before the real test and result in a huge time gain. Running an experiment in a time discrete event simulator such as ns3 can also be faster than running its real time equivalent.

Point 1) alone does not justify yet another implementation since testing can also be realized through alternative means. In simple cases, container-based simulations such as Mininet testbeds can be enough but at higher throughput, hardware limits (e.g. processor speed) can spoil the results: switching to discrete time solution such as DCE (see 3.1) makes sense in that case.

The point 2) can be considered as the stronger motivation, 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 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, developing a simulation model beforehand is reasonably faster and can help realize problems ahead of time.

As a side effect, 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

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.

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 MPTCP's 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

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. 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 established a list of the 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 reducing the window of a subflow that blocks the MPTCP window and the opportunistic retransmission feature.

Also contrary to linux 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 in advance. Creating

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.

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 version 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) and December 2013 for [29] (ns-3.19). This gap in versions make the practical evaluation a challenge as well as the interpretation of results as the ns-3 TCP implementation evolved a lot 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. Tools used

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.

```
    Window size value: 909
    [Calculated window size: 116352]
    [Window size scaling factor: 128]
  ▶ Checksum: 0x6e8b [unchecked, not all data available]
    Urgent pointer: 0
  ▼ Options: (32 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps, Multipath TCP
    ▶ No-Operation (NOP)
    ▶ No-Operation (NOP)
    ▶ Timestamps: TSval 1389505775, TSecr 21868216
    ▼ Multipath TCP: Data Sequence Signal
        Kind: Multipath TCP (30)
        Length: 20
        0010 .... = Multipath TCP subtype: Data Sequence Signal (2)
      ▼ Multipath TCP flags: 0x05
          ...0 .... = DATA_FIN: 0
          .... 0... = Data Sequence Number is 8 octets: 0
          .... .1.. = Data Sequence Number, Subflow Sequence Number, Data-level Length, Checksum present:
          .... ..0. = Data ACK is 8 octets: 0
          .... ...1 = Data ACK is present: 1
        Original MPTCP Data ACK (32 bits): 2020799161
        [Multipath TCP Data ACK: 376 (Relative)]
        Data Sequence Number: 1671049916  (32bits version)
        Subflow Sequence Number: 467
        Data-level Length: 245
        [Data Sequence Number: 467 (Relative)]
  ▶ [SEQ/ACK analysis]
  ▼ [MPTCP analysis]
      [Master flow: master is tcp stream 0]
      [Stream index: 0]
      [TCP subflow stream id(s): 2 1 0]
      [Segment Data Sequence Number start: 1671049916 (64bits)]
      [Segment Data Sequence Number end: 1671050160 (64bits)]
      1671049915 found in packet 16 (current frame=19)
      Application latency: 0.297393000 seconds
```

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

- 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] that leverages these results to produce the plots presented in the next section.

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

We present in the following a few simulations to compare the linux kernel running in DCE 1.7 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 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.

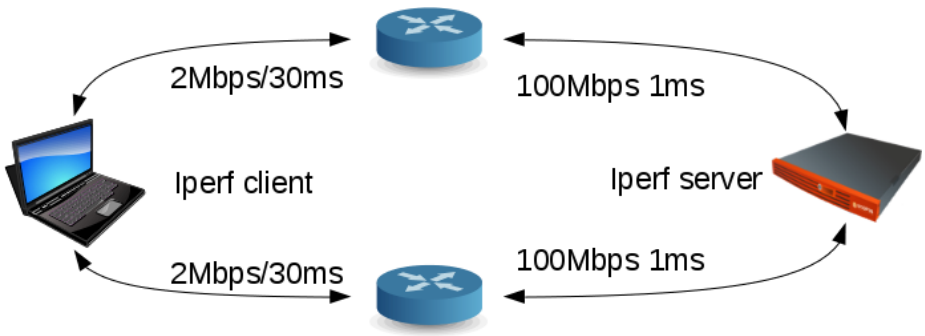


Figure 6: The topology used for the simulations. First hop is the 1Mbps bottleneck, with a variable propagation delay.

The Bandwidth Delay-Product (BDP) refers to the number of unacknowledged bytes that can be inflight. It is generally advised to set the glsbdp higher than $RTT *$ 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 network latency impact the RTT. On one path with a bottleneck of 2Mbps and a RTT of 60ms, the glsbdp is 120kbits. We run the experiments with "libos" [25] applied against the linux mptcp kernel v0.89. Moreover:

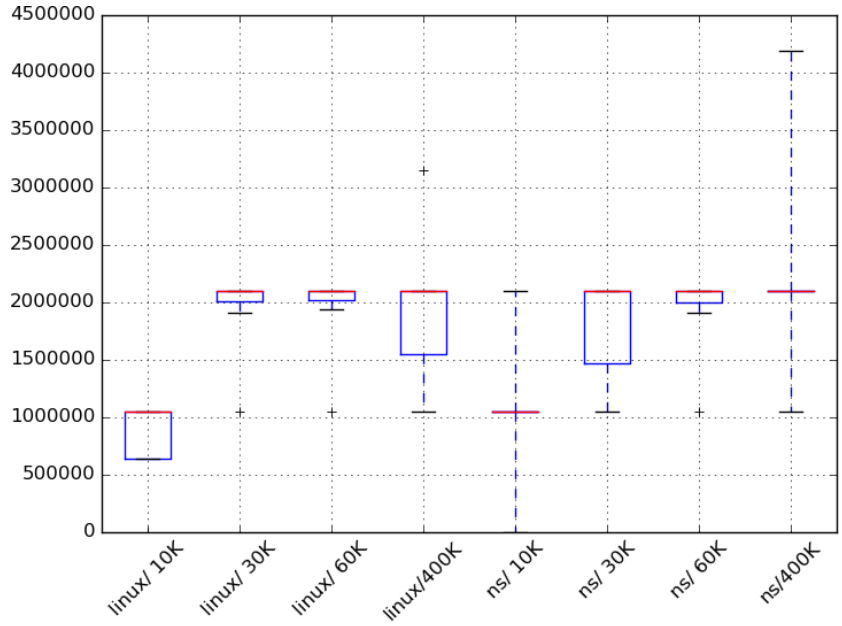
- Scheduler is set to round robin.
- Number of paths is set to one (Figure 7a), then two (Figure 7b).
- Forward and backward one way delays are set to 30ms on each path.
- We launch several runs with different receiver windows.

We ran five seconds iperf2 [9] sessions between the two hosts without any background traffic on the topology of Figure 6.

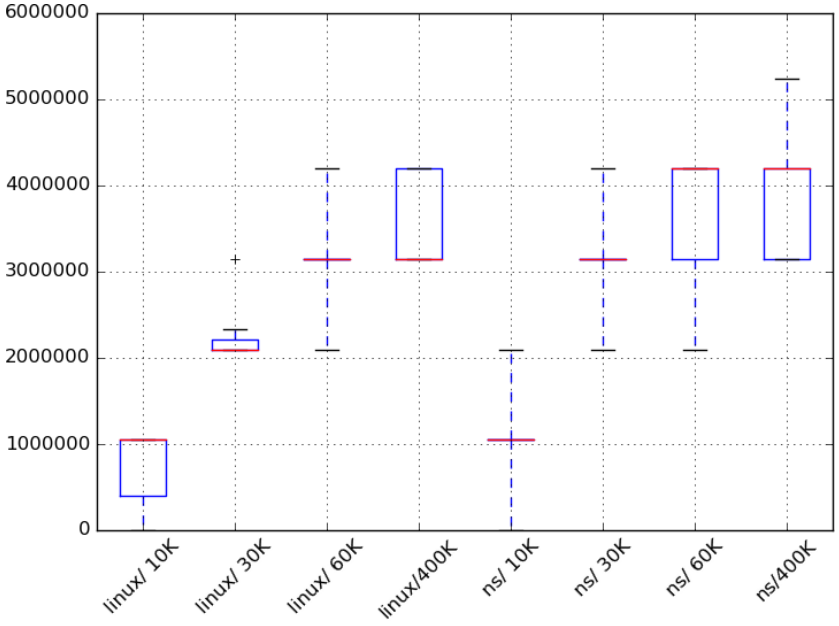
The size of the router buffers is the default linux one.

On Figure 7a, we notice that both stacks make the maximum use of the paths except when it is window limited as for the 10KB case. We can also

[9] http://iperf.sourceforge.net/

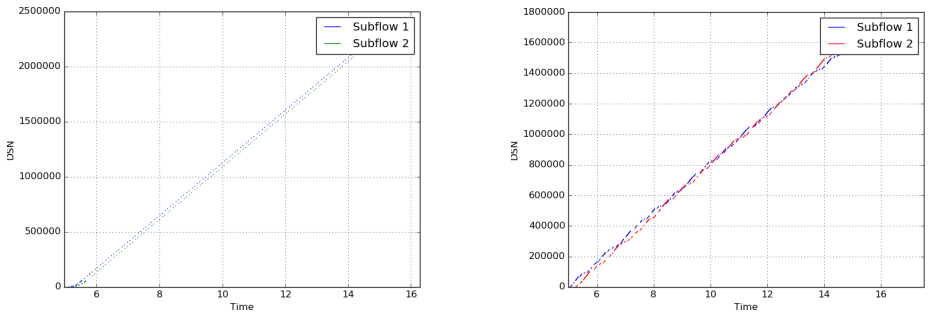


(a) Single path.



(b) Two paths.

Figure 7: MPTCP linux kernel and ns3 iperf2 throughputs.



(a) In ns3.

(b) In linux.

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

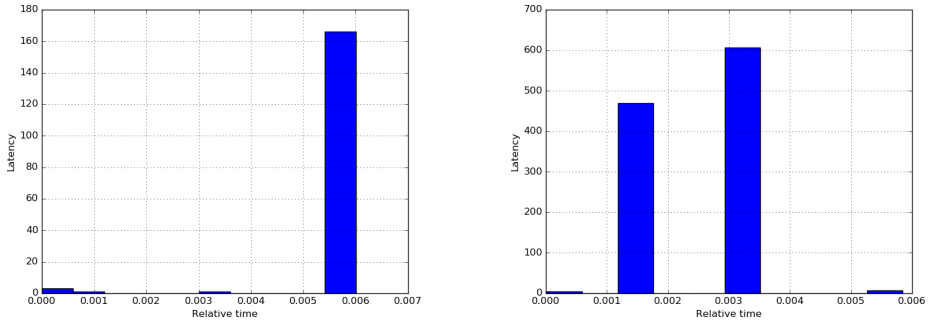
notice that the throughput is a little more than the maximum throughput. We believe it is due to iperf2. Compared to the one path case, we can the expected doubling in throughput in the two paths case on Figure 7b when we add a path when the window is big enough. It also seems that the ns3 version is greedier as in the 30KB window.

In order to check the behavior of the scheduler and thanks to mptcpanalyzer [18], we were able to plot the relative MPTCP sequence numbers transmitted on every subflow for a 40KB 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.

Finally we plot the delay between the arrival of consecutive bytes for both the ns3 (Figure 9b) and the linux cases (Figure 9a). Out of order arrival would be noted as negative values but there is none in this setup. We note that the order of magnitude are similar between the stacks but that the ns3 stack result is bimodal. We suppose it is due to an implementation detail on how events are scheduled..

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



(a) In linux.

(b) In ns3.

Figure 9: Interarrival DSN latencies in a two path network with the round robin scheduler.

implementations and could impact the simulation fidelity in tight buffer simulations. One promising solution is Non-Renegotiable Selective Acknowledgements (NR-SACK) [32]; sadly the source is not available and this would require ns3 to implement SACK first.

Future work. Paasch et al. made an important contribution in applying experimental design [33] 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 in mininet. Network coding is an active area of research, which could improve MPTCP characteristics [34]. While operating system seem to remain oblivious to network coding, there exists a detailed library for ns3 [10].

5. Conclusion

We presented the MPTCP protocol and its new implementation in the network simulator ns3 that conforms to many of the features described by the standard. We hope our effort will allow to develop new schemes in an easier way to improve or find new ways of using a multipath communication. Indeed MPTCP represents a subset of how multipath protocols could improve our future communications and may represent a turning point between TCP

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

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.

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 *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 219–234. [Online]. Available: http://dl.acm.org/citation.cfm?id=2789770.2789786

[2] S. Ferlin, T. Dreibholz, and O. Alay, "Multi-path transport over heterogeneous wireless networks: Does it really pay off?" in *Global Communications Conference (GLOBECOM), 2014 IEEE*, Dec 2014, pp. 4807–4813.

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

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

[5] R. Jesup, S. Loreto, and M. Tuexen, "WebRTC Data Channels," 2015.

[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. [Online]. Available: http://doi.acm.org/10.1145/1452335.1452342

[8] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar, "RFC6182 Architectural guidelines for Multipath TCP Development," *RFC6182*, 2011.

[9] M. Scharf and A. Ford, "RFC6897 - Multipath TCP Application Interface Considerations," 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 *Cloud Networking (CloudNet), 2013 IEEE 2nd International Conference on*, Nov 2013, pp. 58–66.

[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 *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, Nov 2012, pp. 1617–1624.

[12] S. Hassayoun, J. Iyengar, and D. Ros, "Dynamic window coupling for multipath congestion control," in *2011 19th IEEE International Conference on Network Protocols*, Oct 2011, pp. 341–352.

[13] S. Ferlin, A. Alay, D. A. Hayes, T. Dreibholz, and M. Welzl, "Revisiting Congestion Control for Multipath TCP with Shared Bottleneck Detection," in *Proceedings of the 35th IEEE International Conference on Computer Communications (INFOCOM)*, San Francisco, California/U.S.A., Apr. 2016, pp. 2419–2427, ISBN 978-1-4673-9953-1. [Online]. Available: https://www.simula.no/file/infocom2016-webpdf/download

[14] H. Oda, H. Hisamatsu, and H. Noborio, "Design and evaluation of hybrid congestion control mechanism for video streaming," in *Computer and In-*

formation Technology (CIT), 2011 IEEE 11th International Conference on, Aug 2011, pp. 585–590.

[15] C. Paasch, "Improving Multipath TCP," Ph.D. dissertation, 2014.

[16] M. Li, A. Lukyanenko, S. Tarkoma, and A. Yla-Jaaski, "The delayed ack evolution in mptcp," in *Global Communications Conference (GLOBECOM), 2013 IEEE*, Dec 2013, pp. 2282–2288.

[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] M. Coudron, "MPTCP simulator source." [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] "The Application Layer Traffic Optimization (ALTO)." [Online]. Available: http://datatracker.ietf.org/wg/alto/charter/

[21] M. Scharf, G. Wilfong, and L. Zhang, "Sparsifying network topologies for application guidance," in *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, May 2015, pp. 234–242.

[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 *Advanced Information Networking and Applications Workshops (WAINA), 2014 28th International Conference on*, May 2014, pp. 765–771.

[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 *The 9th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Santa Barbara, United States, Dec. 2013. [Online]. Available: https://hal.inria.fr/hal-00880870

[27] M. Coudron, S. Secci, and G. Pujolle, "Differentiated pacing on multiple paths to improve one-way delay estimations," in *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, May 2015, pp. 672–678.

[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 *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 361–362. [Online]. Available: http://doi.acm.org/10.1145/2619239.2631453

[31] M. Coudron, "mptcpanalyzer: 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 *Advanced Information Networking and Applications Work-*

shops (WAINA), 2013 27th International Conference on, March 2013, pp. 1113–1118.

[33] C. Paasch, R. Khalili, and O. Bonaventure, "On the benefits of applying experimental design to improve multipath tcp," in *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '13. New York, NY, USA: ACM, 2013, pp. 393–398. [Online]. Available: http://doi.acm.org/10.1145/2535372.2535403

[34] M. Li, A. Lukyanenko, and Y. Cui, "Network coding based multipath tcp," in *Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on*, March 2012, pp. 25–30.