Analyzing and Realizing Multipath TCP Schedulers in Linux

Hendrik Cech
hendrik.cech@tum.com
Technical University of Munich

ABSTRACT

Today's devices are often equipped with multiple network interfaces. Multipath TCP (MPTCP) provides the opportunity to utilize the capabilities of all available flows for a single logical connection. Because of its huge impact on MPTCP's performance, packet scheduling has been an active research area. Many approaches have been published that aim to improve MPTCP's latency or its behavior given heterogeneous paths. The performance of these solutions can however not easily be compared due to the fragmentation of the existing implementations.

We performed the necessary work to compare a number of promising scheduling solutions. First, we unified the implementations of minRTT [26], roundrobin [26], MuSher [29], BLEST [9], ECF [22], and STTF [14], by bringing them to the same Linux MPTCP [6] version. Then, we setup a testbed based on two Linux machines and two routers that is able to emulate different network scenarios over a variable number of paths. The results obtained from the conducted tests allow the direct and fair comparison of the published scheduling approaches. Key findings are the profound impact of the network queue sizes on the schedulers' behavior and the disparate performances given delay heterogeneous paths.

Subsequently, we analyze the characteristics of LTE networks and sketch a scheduling solution to improve MPTCP's performance in this environment. A focus is on handover detection and recovery. We present benchmarks from a preliminary implementation that attest a good baseline performance.

1 INTRODUCTION

Today's computing devices are often equipped with multiple network interfaces: smartphones are able to connect to WiFi networks and simultaneously maintain one or even multiple LTE connections. Servers in data centers are connected by multiple wired paths to increase the network's fault tolerance. The TCP/IP stack however is only designed for single path connections.

Multipath TCP (MPTCP) is a recent extension of TCP that allows the communication between two end-hosts over multiple paths [10]. One promising application area are mobile scenarios where the available data connections can be used in conjunction to increase the total throughput and the resilience against temporary link failures [25]. Another one are data centers where multipath transport leads to a higher throughput and better load balancing [27]. The broadest production deployment of MPTCP is likely managed by Apple in their iOS devices where MPTCP improves the user experience of their digital voice assistant [4].

The most capable open-source implementation is based on the Linux kernel [6]; it is currently rewritten to be eventually included in the official Linux repository [23]. In this implementation, the use of MPTCP is transparent to the application which needs no changes to take advantage of multipath transport. Similarly, MPTCP is transparent to the network which continues to serve regular single-path TCP (SPTCP) connections.

Despite these promising applications, MPTCP struggles to effectively utilize the available paths if their characteristics in terms of round-trip time (RTT) or bandwidth differ [9]. Similarly, MPTCP does not yet fully exploit its potential for latency-sensitive applications [11].

Packet scheduling has a huge impact on the performance of MPTCP especially in challenging environments [1]. Its core task is to decide over which flow a packet shall be sent. Wrong decisions can lead to head-of-line blocking or prohibit further transmissions due to receive window limitations [9].

Over the last years many approaches to different application areas have been proposed (see Table 1). It is however hard to identify the best performing approaches as the evaluation strategies differ widely. Some publications chose to simulate their scheduler's behavior, while others have based their evaluation on a Linux implementation; but even then only seldom on the same version. Furthermore, the test settings are often not directly comparable.

We set out to provide a fair comparison of published schedulers to gauge their performance in different scenarios and to present an informed view about the current state of research.

The decision about the type of test environment entails a trade-off between flexibility and the closeness to reality. On the one hand, network simulators such as NS-3 provide great flexibility with regards to the environment that the code is executed in. However, the scope of simulations is limited and the obtained results can only be applied with care to the real world. Field measurements on the other hand are cumbersome to conduct and are hard to reproduce. They are however closest to the actual application area of the subject.

We identify a middle ground between those two extremes and implement a testbed that involves multiple physical machines which execute the Linux MPTCP kernel. The testbed is able to emulate different network scenarios over a variable number of paths.

To attribute differences in the measurements results solely to the tested scheduler, we gather published scheduler implementations for the Linux kernel and bring them to a common framework version. Using this methodology, we are able to fairly compare the minRTT [26], roundrobin [26], MuSher [29], BLEST [9], ECF [22], and STTF [14] schedulers.

In addition to the scheduler comparison, we describe the design of an MPTCP scheduler for the requirements of LTE environments which is in particular the detection of and recovery from *handovers*. Preliminary benchmarks show its promising performance.

The rest of this document is structured as follows. Section 2 provides an overview over the scheduler literature and motivates the need for our work. Section 3 introduces the MPTCP stack and describes the idea behind the algorithms that we examine. Section 4

describes our testbed and the testing methodology, before presenting and discussing our results. Section 5 highlights the challenges MPTCP schedulers face in an LTE environment and outlines a novel solution to these problems. Section 6 discusses a selection of problems that we encountered. Finally, section 7 draws a conclusion.

2 RELATED WORK

A lot of research effort was already directed at improving the MPTCP scheduling performance. A central motivation is the poor performance of the default minRTT scheduler given that the performance characteristics (in terms of delay, loss, bandwidth, etc.) of the subflows are different ("heterogeneous networks") [18]. The proposed scheduling solutions are either general-purpose or application-specific. Table 1 provides a non-exhaustive overview.

In the literature, the performance of MPTCP schedulers is evaluated differently. If no implementation is provided, the core idea is at least theoretically verified (e.g., [7]) or implemented for a network simulator (e.g., NS-2/3 as used by [20]/[30]). The most powerful option is to implement the scheduler for the Linux kernel [6, 26] as it can then be tested in the wild. An interesting alternative to developing the scheduler directly in C is to use the ProgMP [12] framework that executes schedulers in the Linux kernel that are defined in a domain-specific language.

The evaluation of most publications that propose a new scheduler are limited in one way or another. Often the breadth of other schedulers that were subject to their tests is limited to the built-in minRTT, roundrobin and redundant schedulers (e.g., [11]). If schedulers proposed in the literature are included, it is often unclear if their Linux implementations are based on the same kernel version (e.g., [8, 14, 22]). If they are not, performance differences can not necessarily be solely attributed to the scheduling algorithm.

Kimura et al. set up a controlled testbed to simulate different bottleneck characteristics and conducted systematic tests that benchmarked pairs of scheduling strategies and MP congestion control algorithms. Their test procedure is very thorough and unique because it examines the interplay of schedulers and congestion control mechanisms. The selected schedulers are however not taken from the literature but algorithms that are based on a single critertion (e.g., the available sending window space) [17, 18].

Finally, it is often impossible to fairly compare the evaluation results of different papers due to the differences in their implementations, their test environment and their test methodology. To arrive at a meaningful comparison of MPTCP schedulers, we first moved all scheduler implementations onto the same version of the Linux kernel and the MPTCP framework. Next, we set up a testbed that evades a lot of the pitfalls of simulated setups. In our testbed, the sending and the receiving parts are on two different physical machines. Similarly, network shaping was applied on physically-separated routers that control all traffic between the interacting machines. Given those conditions, we were able to repeatedly conduct the same tests with all schedulers and draw meaningful conclusions.

3 MULTIPATH TCP SCHEDULERS

The MPTCP Linux project [6] maintains the most popular open source MPTCP implementation that integrates the MPTCP protocol into the Linux kernel. Our work is based on MPTCP v0.94 from March 2018 that is bundled with kernel version 4.14. The logical structure of an MPTCP connection as it is integrated into the Linux kernel is shown by Figure 1.

The setup of an MPTCP connection happens transparently to the application which does not need to be modified but can continue to use the kernel socket interface. The application interacts with the MPTCP meta socket that manages the connection-level send buffer and assigns data a connection-level sequence number. The path manager is responsible for setting up regular TCP connections over which data is transported ("subflows"). Multiple strategies are available; the most common one is *fullmesh* which attempts to establish connections between all combinations of local and remote endpoints.

The congestion control of the individual subflows is coupled to be fair to concurrent TCP connections and reach a higher performance. Four algorithms are available: LIA [28], OLIA [16], BALIA [33], and wVegas [35]. A concise summary of those approaches are given by [19].

The task of deciding which data shall be sent on which subflow is assigned to the *scheduler*. The MPTCP framework invokes it whenever space is available in the connection-level send window or when a retransmission is triggered on the meta level. Most schedulers base their decisions upon metrics provided by the TCP network layer such as the state of the flows' congestion windows and RTT estimates. Some however also use information from other layers. QAware [30] for example directly interacts with a lower layer and retrieves information about the queue state from the NIC driver. The "Cross-Layer Scheduler for Video Streaming" [7] on the other hand retrieves information about the type of data that is encapsulated in a packet from the application layer.

An scheduler implementation for the Linux kernel needs to implement two functions that are called by the MPTCP framework at the appropriate times. The function `next_segment` is expected to select a segment and a subflow on which the MPTCP framework will schedule that segment. Most often, the segment will be taken from the MPTCP meta send queue but a second source is the reinject queue from which a scheduler is expected to read with priority. Segments that were scheduled on a particular flow but could not be sent are moved to the reinject queue such that they undergo the flow assignment process again. One reason can be that a subflow is closed; unsent packets are moved to the reinject queue such that they are sent on another subflow.

The function `get_subflow` receives a segment and is expected to return a subflow on which that segment shall be sent. Most often, `get_subflow` is invoked by `next_segment` but the framework also calls this function directly if a particular segment shall be retransmitted.

The MPTCP source files are based in directory `net/mptcp` and its headers in `include/net/mptcp.h`. The implementation is not entirely self-contained; the Linux network stack and in particular its TCP implementation is augmented in various places with MPTCP-specific code. A central mechanism that is used to inject MPTCP specific behaviour is the `tcp_sock_ops` struct that holds function pointers for various tasks. A MPTCP connection has one set of callbacks defined for the meta socket and one set for the

Table 1: MPTCP schedulers

Name	References	Application area	Core idea
minRTT	v0.94 (in-tree), [26], Section 3.1	Heterogeneous networks	Assigns packets to the subflow with the smallest RTT among the subflows that are not CWND-limited.
roundrobin	v0.94 (in-tree), [26], Section 3.2	Research	Picks one flow after the other when assigning packets but skips unavailable ones. For bulk-transfers, the scheduling decision becomes ACK-clocked.
redundant	v0.94 (in-tree)	Research, Low-latency	Sends each segment on all available flows. Can not exceed the goodput of single-path TCP.
DAPS	v0.89[1], 2014, [20]	Wireless networks	Its goal is to reduce receiver buffer blocking by increasing the in-order arrival at the receiver. DAPS generates a sending schedule based on the paths' RTTs.
OTIAS (Out-of-Order Transmission for in order arrival)	ns-2, 2014, [36]	Real-time applications	OTIAS assigns a packet to the flow where it estimates that the packet will arrive soonest. The scheduler appends packets to flows whose CWND is currently full as they will be sent as soon as space is available again.
BLEST (Blocking Estimation)	v0.95 (in-tree), 2016, [9], Section 3.4	Heterogeneous networks	Tries to avoid head-of-line blocking at the receiver by optimizing the occupation of the MPTCP send window. BLEST may skip sending on a slower subflow if that would prevent the faster subflow from sending once it is available again.
Cross-Layer Scheduler for Video Streaming	no implementation available, 2016, [7]	Video streaming	Improve the playback quality by only sending those video units that are likely to arrive in time to be played. Couples information about the data that is being sent with knowledge about the network ("cross-layer approach").
ECF (Earliest completion first)	v0.89, 2017, [22], Section 3.5	Heterogeneous networks	Assigns a packet to the subflow over which it will arrive soonest. To achieve this, ECF might decide to not utilize a slower subflow and instead wait for a faster subflow to become available again.
Scheduling for Thin Streams	2018, [11], progMP implementation	Long running, low throughput, latency sensitive connections	The scheduler builds upon minRTT by actively refreshing the delay estimates of the slower and therefore unused subflows. In addition, it estimates the one-way delay instead of assuming it to be equal to RTT/2.
QAware	v0.93, 2018, [30]	Bulk data transfers	Uses the occupancy of the NIC queues together with RTT estimates to achieve its goal of increasing the overall throughput.
MuSher (Agile MPTCP scheduler)	v0.94, 2019, [29], Section 3.3	Dual WiFi setups	Sets the packet assignment ratio to the throughput ratio of the flows. The scheduler is designed to target the challenges of a dual 802.11ac / 802.11ad setup.
STTF (Shortest Transfer Time First)	0.91.2, 2019, [14], Section 3.6	Heterogenous networks	STTF assigns all unsent segments to subflows, based on the shortest predicted transfer time. This rescheduling is redone on every interruption, e.g., on every ACK arrival.

TCP subflows. The same mechanism is used to allow the implementation of different schedulers: Each scheduler registers itself with an instance of the `mptcp_sched_ops` struct that holds function pointers.

3.1 minRTT (the "default" scheduler)

The default scheduler of the Linux MPTCP implementation schedules packets to the subflow with the lowest RTT and space in its congestion window (CWND). If a flow is available, it will be used to send a queued segment; the scheduler will not *proactively* decide to leave a flow idle. minRTT's mechanism to limit the effect of out-of-order delivery and head-of-line (HoL) blocking is *opportunistic retransmission and penaliztion*. The scheduler retransmits packets on a faster flow if head-of-line blocking on MPTCP-level is experienced and reduces the CWND of the flow that triggered the HoL block. This mechanism is commonly classified as *reactive* [26].

3.2 Roundrobin

The roundrobin scheduler picks one flow after the other when assigning packets without preferring, e.g, flows with a smaller RTT. It does however skip flows whose send or congestion window currently prohibits sending. During bulk transfers (i.e., the send queue is backlogged) the scheduler will become *ack-clocked* and not realize a true roundrobin packet assignment. In this scenario, each subflow is limited by its CWND. Once an ACK arrives on a certain flow, the scheduler will assign the next segment to this flow as it is the only available option [26].

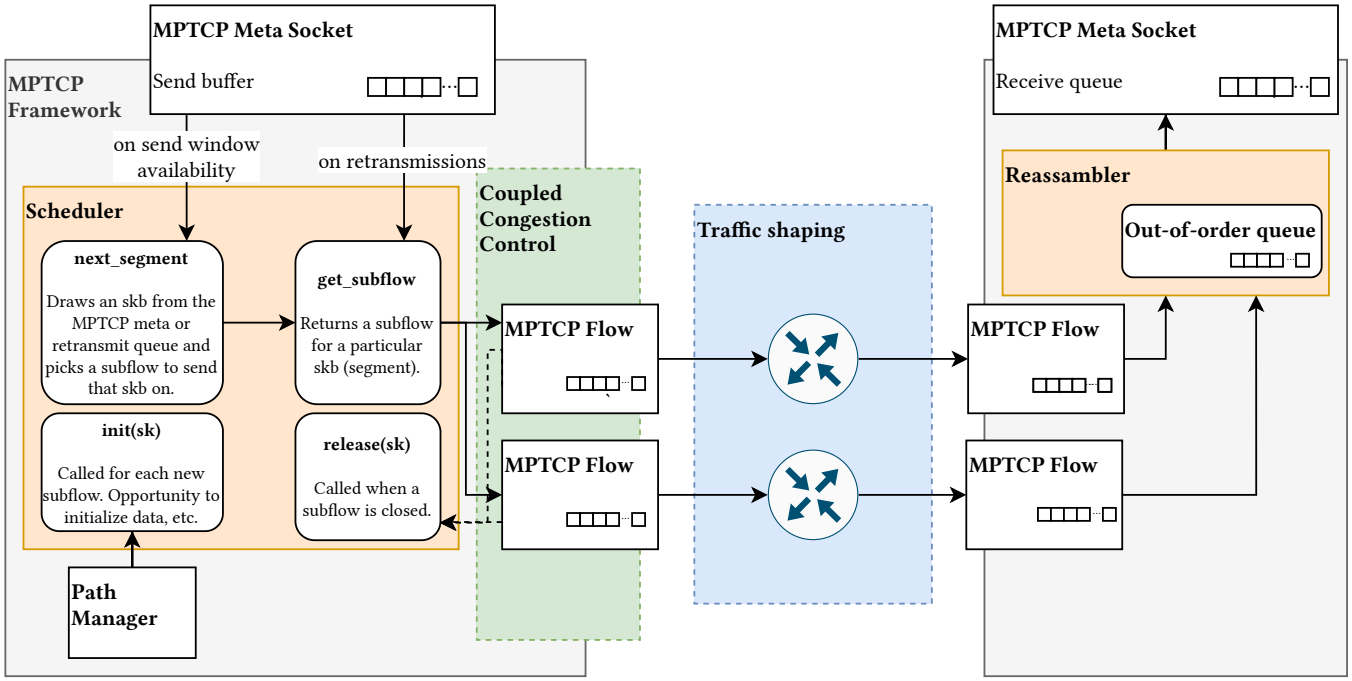

Figure 1: Schematic depiction of the Linux MPTCP components on the sending and the receiving sides. The setup equals our testbed: the connected parties communicate over two paths that include a router which is able to emulate different network scenarios.

3.3 MuSher (Agile MPTCP Scheduler)

MuSher is designed for the requirements of 802.11ac / 802.11ad dual WiFi setups, e.g., frequent recovery from link failures. Their central finding is that optimal MPTCP performance is obtained if the packet-assignment ratio matches the throughput ratio of the two subflows. To do so, the scheduler refreshes its estimate of the path bandwidths whenever the total MPTCP throughput decreases or the send-queue occupancy of any flow decreases. The optimal ratio is found by iteratively changing the ratio and observing the throughput. The search is started into the direction where a throughput increase is detected and stopped once the total throughput drops again [29].

3.4 BLEST (Blocking Estimation)

Algorithm 1: BLEST scheduling algorithm (taken and adapted from [14, Algorithm 1])

1 **if** *fastest subflow (i.e., smallest RTT)* x_f *is available* **then**
2 **return** x_f
3 **else if** *slower subflow* x_s *is available* **then**
4 $\text{rtts} = \text{RTT}_s/\text{RTT}_f$
5 $X = \text{MSS}_f \cdot (\text{CWND}_f + (\text{rtts} - 1)/2) \cdot \text{rtts}$
6 $Y = |\text{MPTCP send window}| - \text{MSS}_s \cdot (\text{inflight}_s + 1)$
7 **if** $X \cdot \lambda > Y$ **then**
8 **return** x_s
9 **return** *no available flow*

BLEST is designed to increase MPTCP's performance over heterogeneous paths. The authors identified head-of-line (HoL) blocking on the MPTCP meta level as the main cause for performance issues in this setting. BLEST monitors the MPTCP send window to reduce the times where the faster subflow can not send a packet due to insufficient space. Similar to minRTT, BLEST schedules packets to the fastest (lowest RTT) subflow if available. If not, it wagers if sending the segment on a slower subflow would presumably block the faster flow from sending once its available again. This would happen if no space in MPTCP's send window is available at that later point. The algorithm involves a scaling factor λ that is tuned during execution based on the accuracy of the scheduling decisions [9, 14].

3.5 ECF (Earliest Completion First)

ECF aims to increase the utilization of the faster of multiple heterogeneous paths. It takes the subflow's RTT estimates, their congestion window occupation, and the size of their send buffers into account. Algorithm 2 describes the scheduling algorithm for two subflows. If available, ECF uses the fastest subflow, i.e., the one with the smallest RTT. Otherwise, it estimates if waiting for the faster subflow to become available would result in a faster completion than using the slower subflow right now (line 7). The second condition (line 8) verifies the same question but by looking at the CWND usage. The scheduler adds some hysteresis to its scheduling decision by setting the variable `waiting` to slow the rate of switching between the available flows [22].

Algorithm 2: ECF scheduling algorithm (taken and adapted from [22, Algorithm 1])

if *fastest subflow (i.e., smallest RTT)* x_f *is available* **then**
  return x_f
else
  Select x_s using the default minRTT scheduler
  $n = 1 + \frac{k}{\text{CWND}_f}$
  $\delta = \max(\sigma_f, \sigma_s)$ `// σ denotes the RTT variation`
  if $n \cdot RTT_f < (1 + waiting \cdot \beta)(RTT_s + \delta)$ **then**
    if $\frac{k}{CWND_s} \cdot RTT_s \geq 2 \cdot RTT_f + \delta$ **then**
      waiting = 1
      return *no available subflow*
    return x_s
  waiting = 0
  return x_s

3.6 STTF (Shortest Transfer Time First)

Algorithm 3: STTF scheduling algorithm (taken and adapted from [14, Algorithm 2 and 3])

remove all packets on all subflow for rescheduling
for *each unsent segment p* **do**
  for *each available subflow s* **do**
    T_s^p = `TransferTime(`s, p`)`
  assign p to s with smallest T_s^p
function `TransferTime(`s, p`)`:
  if *cwnd_free* > 0 **and** *data_to_send* < *cwnd_free* **then**
    return *rtt* /2
  transfer_time += rtt
  cwnd = increase_cwnd(current_cc_state)
  if *data_to_send* ≤ *max_segments_in_ss* **then**
    transfer_time += rtt · (round_in_ss - 1) + rtt/2
    return *transfer_time*
  else
    if *cwnd* < *ssthresh* **then**
      transfer_time += max_rounds_in_ss · rtt
      if *ends_in_ss(data_to_send)* **then**
        return *transfer_time*
      cwnd = ssthresh
    transfer_time += rtt · (rounds_in_ca - 1) + rtt /2
  return *transfer_time*

STTF assigns packets to the flow which it expects to deliver the packet to the receiver in the shortest time. The estimation takes the current congestion control state into account to predict the increase of the CWND during the current RTT. Using the predicted CWND, STTF calculates when the packet will be sent and arrive (see Algorithm 3) [14].

4 TESTBED DESIGN & MEASUREMENTS

Our testbed is based upon two Linux machines and two routers. One machine is generating traffic which is directed to the other machine. They are connected by two disjoint paths. Each path contains a router that shapes the machines' traffic according to configurable rules. The Linux boxes are NRG Systems IPU443 machines equipped with a dual core Intel i5-4300Y CPU, 8GB of RAM, and 4 Gigabit Ethernet ports. We installed Ubuntu 16.04.6 on the machines as its Kernel version (4.4.0) is closest to the Kernel version of MPTCP release v0.94 (4.14.x) among the Ubuntu LTS releases. The routers are Ubiquiti EdgeRouter Xs which provide 5 Gigabit ports. They were setup with OpenWRT 19.07.3 which provides direct SSH access and great flexibility with regards to the routers' function.

For all MPTCP performance tests, iperf3 was used to generate traffic. During the tests, tcpdump saved the first 88 bytes of each packet to include the header of each TCP packet. In addition to data mined from the network traces, the state of key TCP values from the kernel was captured. To that end, an adapted version of the scheduler and queue probe functionality published by [29] was employed. Those kernel modules are modeled after tcp_probe [32]; they expose a virtual file to which data is written that can be read from user space. We used these probes to capture kernel values such as the congestion window or the out-of-order queue size.

Unless otherwise noted, each test scenario was executed for 60 seconds and repeated 5 times. Parameters such as the congestion control algorithm and the TCP window sizes are left unchanged, i.e., the default values set by MPTCP v0.94 were used. The first 5 seconds of each test execution were excluded from the calculation of metrics to let the configuration reach a stable state.

4.1 Scheduler implementations

Table 1 lists a selection of proposed scheduling strategies. Only some of them developed an implementation for the Linux MPTCP framework and even fewer were published. To compile a fair comparison between those schedulers, we selected version 0.94 of the Linux MPTCP project as the common base for all implementations. The kernel version 4.19 is still supported and some schedulers were already developed for it. In addition, the most recently released version 0.95 appears to require far more porting effort.

As far as possible the schedulers were merged into a single codebase which avoids inconsistencies and improves usability. The sysctl interface can for example be used to quickly switch between the available schedulers. This however brings the risk of introducing interference between the competing implementations. We consolidated implementations of the following schedulers for evaluation.

- minRTT and roundrobin are part of the 0.94 distribution.
- MuSher's implementation was already based on 0.94 and required no changes. The code however provokes kernel panics at irregular times. We were not able to find and fix the root cause but were able carry out all experiments.
- ECF was published for version 0.89 that was based on kernel 3.14. To integrate ECF into the unified codebase, the code needed to be restructured. In addition, changes to the TCP stack had to be taken into account (e.g., replacing `tcp_time_stamp` with `tcp_jiffies32`).

- BLEST was integrated into the upstream project and released as part of v0.95. Only minor changes were necessary to migrate BLEST to v0.94.
- STTF was ported to version 0.94 from 0.91.2. Its implementation depends on changes to the TCP and MPTCP stack which obviously also affects other schedulers. We did therefore not integrate STTF into the unified codebase.

Even though we were able to obtain and compile the DAPS source code as provided by the authors, we were not able to use the scheduler due to kernel panics. As the cause could not be identified and fixed, DAPS was not included in our evaluation.

4.2 Network emulation

The traffic shaping on the middleboxes is realized using Traffic Control's (*tc*) queuing discipline (*qdisc*) features that are built into the Linux kernel. In particular, the *NetEm* ("Network Emulator") module was used to delay and drop packets, and to cap the paths' bandwidths.

A central NetEm parameter that affects all its operating modes is *limit*. It controls the size of the buffer that is used to queue packets while, e.g., an additional delay is emulated or the bandwidth is limited. Its minimum size depends on the network delay and the packet rate. If the buffer is too small, packets are dropped and the desired throughput can not be reached. If the queue has a high capacity, *bufferbloat* [5] can occur.

When NetEm is configured on an interface, only outgoing traffic is affected by its rules. The routers that are used in our testbed contain sender and receiver in their personal VLANs that are exposed as network interfaces. Traffic is bridged between them. Which traffic direction is affected by a NetEm rule therefore depends on the interface that this rule is attached to.

4.3 Number of available paths

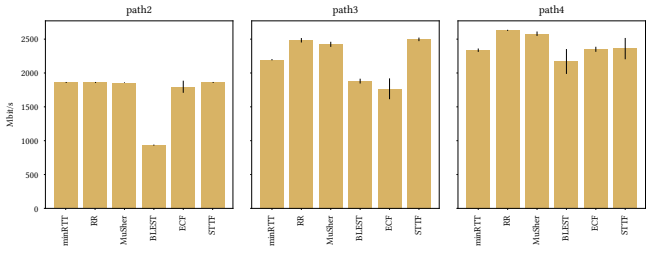

Figure 2: The goodput achieved by different schedulers given two, three, or four Gigabit paths.

MPTCP is designed to work with a variable number of flows. The focus in the literature is however predominantly on the use of MPTCP with 2 flows. While popular end-user applications indeed work with two flows (e.g., a smartphone that concurrently maintains an LTE and a WiFi connection), other applications commonly make use of more flows (e.g., server to server connections in data centers) [3].

Figure 2 shows the goodput of a MPTCP connection between a sender and a receiver that are connected point-to-point by 2, 3, and 4 Gigabit Ethernet cables (i.e., no in-between router). All schedulers are able to almost completely utilize the available bandwidth of two paths (except BLEST, who only used one path). Given three paths, all schedulers increase their total goodput but only roundrobin, MuSher, and STTF maintain a high path utilization (82%) while BLEST and ECF only utilize 62% of the available bandwidth. Given four paths, each scheduler further increases its total goodput but the difference is smaller than between two and three paths (STTF's average goodput actually decreases by 136 Mbit/s, but the difference to three paths is within the error margin). The bandwidth utilization drops to an average of 60%.

These results challenge the measurements of [22, Figure 15] who compared the bandwidth utilization of minRTT and ECF when using 4 subflows. In their comparison ECF performs better than minRTT, especially with higher bandwidth. Still, the results are not fully comparable as Lim et al. conducted their tests with a maximum path bandwidth of 8.6 Mbit/s.

We verified that the TCP buffers sizes do not limit the throughput and that our hardware is able to fully utilize four Gigabit paths. To achieve this, the number of MPTCP connections had to be scaled; if the receiver runs two iperf processes and the sender uses four processes to generate traffic, a total goodput of 3.8 Gbit/s can be maintained by the minRTT scheduler. This suggests that a code-optimized scheduler and MPTCP stack could increase the throughput of a single connection.

4.4 Heterogenous bandwidths

Figure 3 shows the scheduling performance given paths with heterogeneous bandwidths once with large and once with small queues.

The results show that minRTT and STTF don't perform well if the network queues are large. Not only is their total goodput below average but they drastically increase the out-of-order queue usage and the SRTT compared to the setting with small queues. Once the paths' bandwidths differ, minRTT and STTF fall back to using only the unimpaired path. RR, BLEST, and ECF on the other hand actually show better performance than with small network queues. Their goodput is slightly higher and their OFO queue occupation lower than in the small queue setting.

Overall, the first flow's goodput is unaffected by the reduced bandwidth of the second path. An exception is MuSher which does not detect the paths' asymmetric bandwidth capabilities correctly and schedules an equal number of packets to each flow. Its overall goodput is therefore limited to twice the second path's bandwidth capability.

The measured RTT over the impaired path increases with increased bandwidth heterogeneity. Looking at the unimpaired path, this effect is also triggered by large network queues while the delay does not change in the tests with small queues.

Finally, a greater bandwidth-heterogeneity increases the out-of-order queue size at the receiver. Interestingly, this effect is more pronounced in the tests with small network queues. In this regard our results match [9, Figure 8] who also found a greater out-of-order queue size usage of minRTT compared to BLEST.

4.5 Path delays

Figure 4 depicts the impact of network delay on the scheduling performance. The change of the base delay is clearly visible in

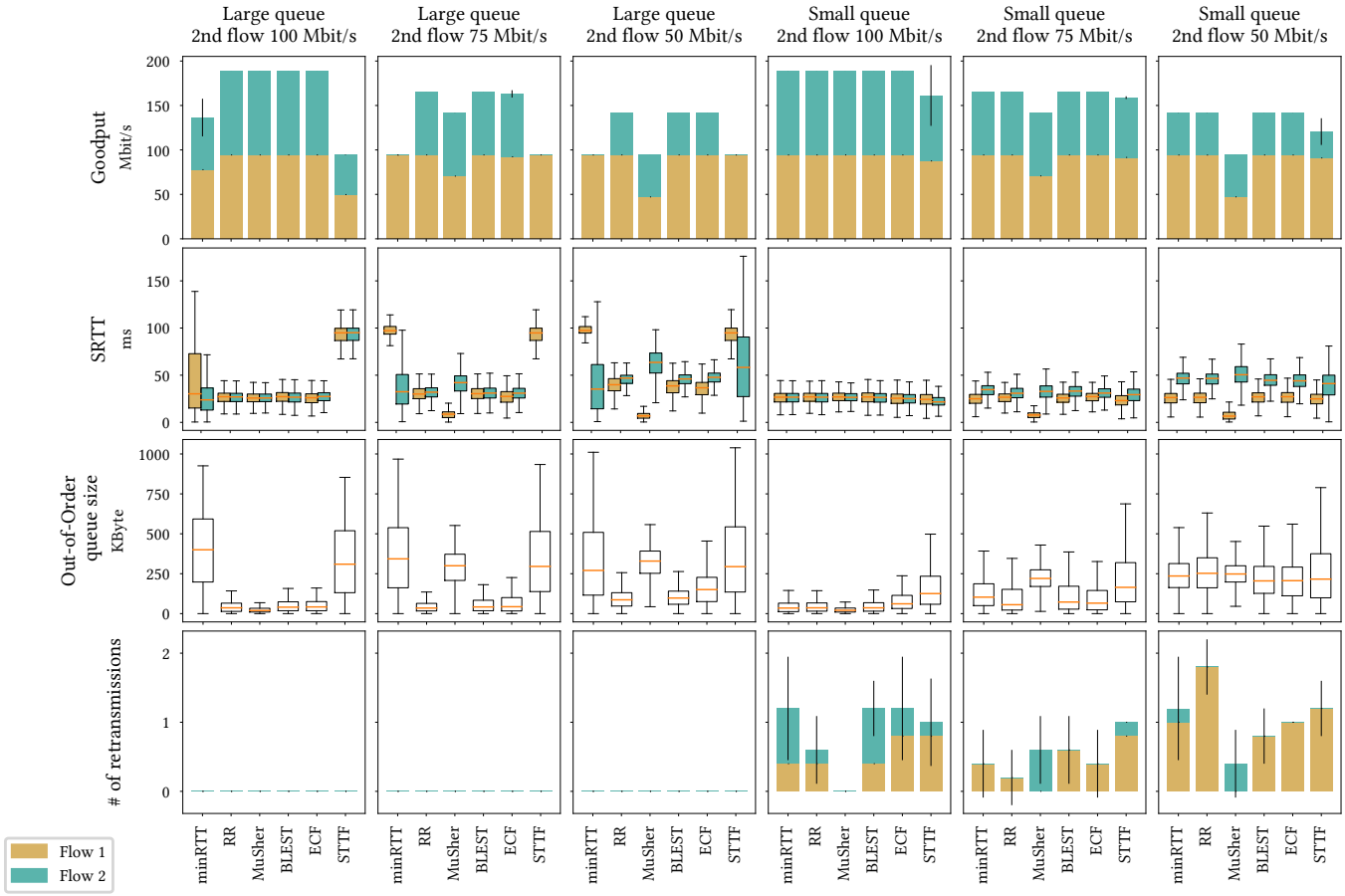

Figure 3: Comparison of the scheduling performance on paths with heterogeneous bandwidths. The bandwidth of the first flow was capped to 100 Mbit/s while the bandwidth of the second flow was reduced iteratively. Each setting was tested with two different queue sizes at the intermediate router: once with "small" queues (300 packets) and once with "large" queues (1000 packets).

all runs which results in an decrease of the overall throughput. If the network provides large queues (controlled by the NetEm limit parameter), the schedulers provide comparable performance while MuSher has a bit of an edge in the low-delay scenario. An outlier is STTF, which shows similar weaknesses as described in Section 4.4.

The situation is flipped given small queues: the goodput of minRTT, RR, BLEST, and ECF shrinks significantly, MuSher's goodput drops to a very low point, and STTF performs bets. While RR, BLEST, and ECF almost exclusively use one flow, minRTT and MuSher are able to utilize both flows equally. This comes at the cost of retransmissions but not at the cost of a high out-of-order (OFO) queue utilization.

In this regard MuSher does exceptionally well and is able to keep its usage low throughout all six tests. With large queues, BLEST, ECF and minRTT yield a high OFO queue usage which is also reflected in the utilization ratio: the higher the OFO queue size, the bigger was the polarization towards one flow. Those schedulers' OFO queue usage drops with small queues; as the total goodput is also lower, the size of this effect is smaller than it seems at first. The opposite effect comes forth with the roundrobin scheduler: its average OFO queue size is the largest.

Figure 5 shows the schedulers' performance on path with heterogeneous delays. With the exception of RR, MuSher, and STTF, the schedulers only utilized one flow.

Considering the resulting goodput and the required OFO queue usage, BLEST performs best across the scenarios by simply deciding to only use one flow. These results don't contradict [9, Figure 8] who found that BLEST at least matched the throughput of the faster path while keeping its OFO queue usage slightly lower than minRTT's. Similarly, [30] benchmarked minRTT, BLEST, and ECF in a comparable scenario and also found that minRTT produces slightly more traffic in delay-homogenous settings but is passed by BLEST and ECF once the delay difference increases.

Hurtig et al. compared the performance of minRTT, BLEST, and STTF over two bandwidth-limited paths where the delay heterogeneity was progressively increased [14, Figure 9]. They found that MPTCP's goodput dropped below single-path TCP's goodput once the delay difference between the two paths got too large. We

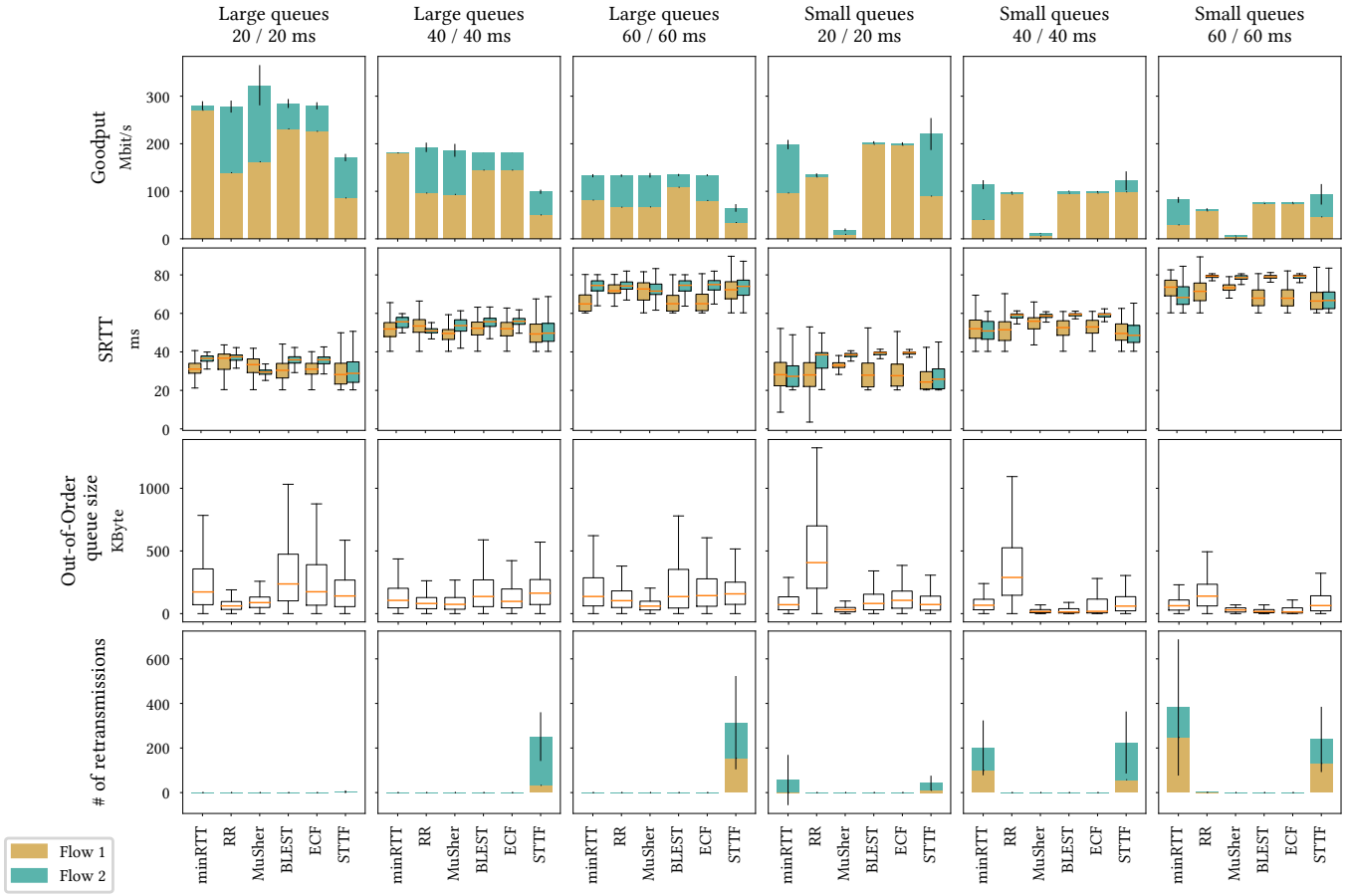

Figure 4: Comparison of different schedulers on paths with increasing delays once with large (2000 packets) and once with small (500 packets) network queues . The delay of both flows was equally increased by 20, 40, and 60ms respectively. In all cases half of the total delay was added to the up-link while the other half was added to the down-link, i.e., data and acknowledgement segments were equally delayed. The bandwidths of the Gigabit-paths were not limited.

can't reproduce this behavior as the goodput stays constant with increasing delay discrepancy; the schedulers decide to ignore the impaired flow rather than to reduce the total goodput by using both subflows.

4.6 Heterogeneous packet loss

All schedulers are able to efficiently deal with a lossy flow as shown by Figure 6. The throughput on the first flow, that experiences only very infrequent packet loss, stays close to its maximum. Only once the loss on the second path is increased to the extremely high value of 5%, the throughput on the first flow takes a small hit as well.

The out-of-order queue size and the number of packet retransmissions increases alongside the loss probability as expected. The measured sRTT of the second flow however drops close to zero once its loss probability is higher than 0.5%. This effect can be attributed to congestion control which keeps the number of in-flight packets on the impaired flow low. The occupancy of the involved queues stays low and the RTT is not noticeably increased above its base level.

These results suggest that the scheduler's response to packet loss is adequate and not a pressing research issue. A simple explanation for their uniform behavior is that they have only little influence on the packet loss response: retransmissions are handled on the individual subflows by the unchanged TCP stack and by the MPTCP framework.

4.7 Influence of congestion control

Figure 7 shows the scheduling performance in combination with the coupled congestion control mechanisms that are available in the MPTCP Linux kernel implementation: LIA [28], OLIA [16], BALIA [33], and wVegas [35]. While LIA, OLIA, and BALIA only react to congestion once packet loss occurs, wVegas tries to estimate the queue delay of bottleneck links to predict congestion and react proactively [19]. The measurement results show that wVegas is not able to work effectively in our test environment. This behavior is surprising as recent measurements showed wVegas' competitive performance in very similar environments [18].

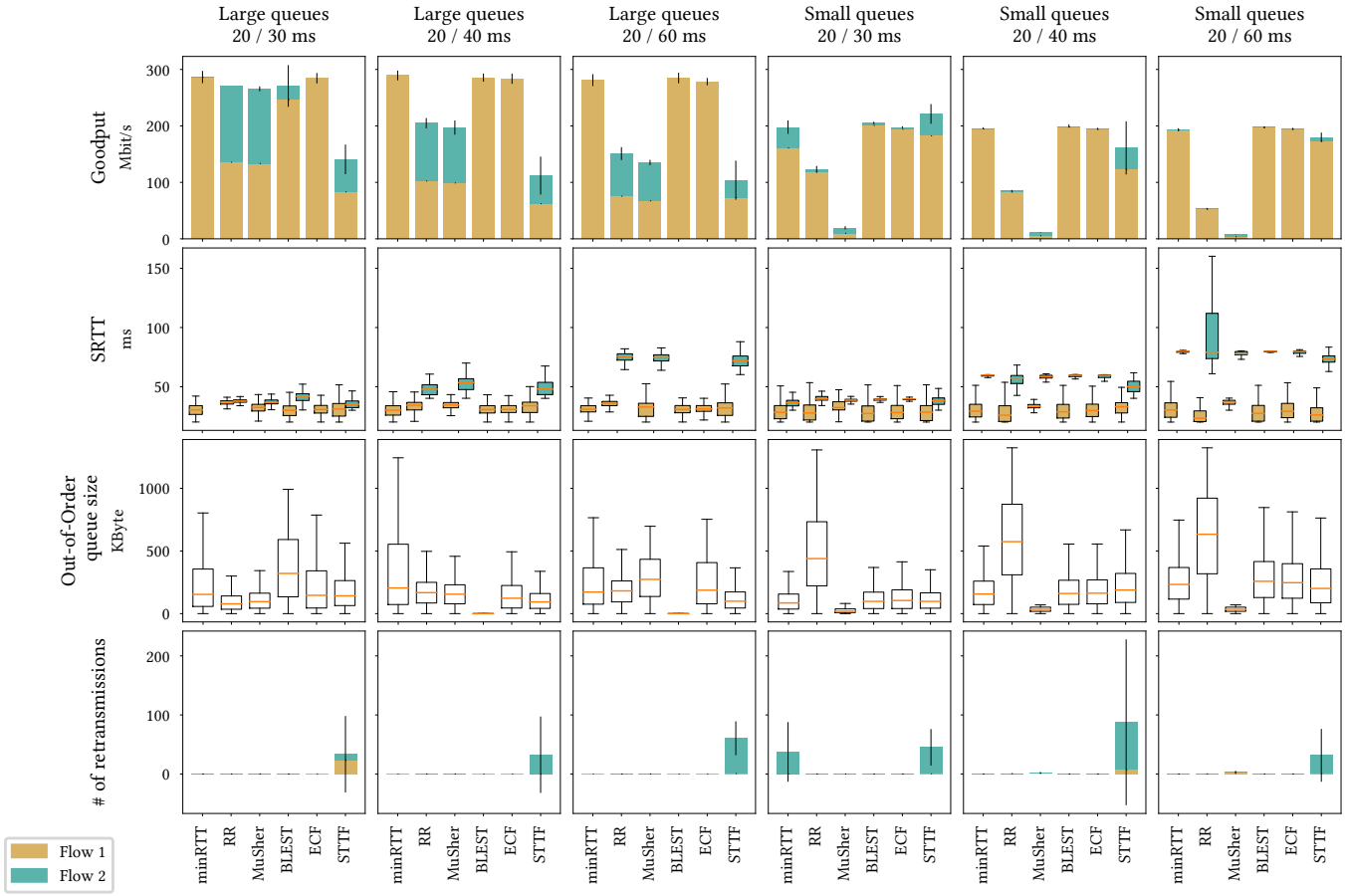

Figure 5: Comparison of different schedulers on paths with heterogeneous delays once with large (2000 packets) and once with small (500 packets) network queues. The delay of the first path was fixed to 20ms while the additional delay of the second path was incrementally increased by 30ms, 40ms, and 50ms respectively. In all cases half of the total delay was added to the up-link while the other half was added to the down-link, i.e., data and acknowledgement segments were equally delayed. The bandwidths of the Gigabit-paths were not limited.

The three similar algorithms LIA, OLIA, and BALIA also show comparable performance. Interestingly, minRTT's persistent problem with large queues does not occur if paired with BALIA. This should be investigated further.

5 DESIGNING AND IMPLEMENTING A SCHEDULER FOR LTE

LTE networks exhibit specific characteristics that were analyzed by [2]. They find that the data capacity of LTE up- and down-links are nearly symmetric and that packet losses happen very rarely and are even then mostly due to congestion in the carrier backlink network. This is achieved by the Hybrid Automatic Repeat Request (HARQ) technique which retransmits erroneous or lost blocks on the link layer.

An inherent behaviour of mobile data connections are *handover* events which happen when the client physically moves and needs to switch from one radio mast to another. Measurements at high speeds show that handovers cause a throughput drop by a factor of 10 while the RTT increases 3-fold [21]. Their duration varies: 85% of successful ("type 1") handovers complete in less than 100ms while the duration increases to more than one or even several seconds during more difficult scenarios ("type 2 and 3") [34]. In this scenario handovers happen frequently (multiple times per minute) but the probability of concurrent handovers on independent carrier networks is very low. This provides an opportunity for MPTCP to compensate the temporary failure of one path by rerouting traffic over another one.

Figure 9 shows the default behavior of MPTCP in an emulated dual LTE environment. Every 15 seconds a handover lasting 2 seconds was emulated on the second subflow (the parameters were taken from [21, 31] as previously described). Strikingly, the throughput of both flows dropped during the handover. The recovery from the handover is delayed.

The main challenge for a scheduler designed for LTE networks is to improve MPTCP's response to handover events. To quickly

Figure 6: Comparison of different schedulers on paths with heterogeneous packet loss probabilities. The up-link bandwidth of both flows was limited to 100 Mbit/s. The packet loss probability of the first path was fixed at 0.005%, while the loss probability of the second path was incrementally increased. The loss was applied equally to the up- and the down-link.

respond and recover from handovers, the scheduler should behave as follows.

(1) If a flow experiences a handover, the scheduler should not push new packets to this flow. To prevent head-of-line blocking on the MPTCP meta level, scheduled and in-flight packets of the affected flow should be proactively resent.
(2) After the handover is completed, the round-trip-time estimate of the flow that is now served by a different radio mast is likely inaccurate. The scheduler should therefore refresh that flow's RTT estimate to improve its scheduling decisions.

To achieve goal (1), the on-setting handover has to be detected as quickly as possible. A promising approach is to monitor the time between packet send (*inter-send*) events and acknowledgement arrivals (*inter-ack*). On an uncongested flow, the inter-send time will closely match the inter-ack time once TCP reaches a steady state because the sender will become ACK-clocked, i.e., will be limited by its send window. An increase of inter-ack time relative to inter-send time would signal a rising path delay which in turn is a typical symptom of an LTE handover.

To achieve goal (2), the scheduler needs to schedule packets on the path so that TCP's smoothed round-trip-time (*sRTT*) calculation can work with relevant data. These packets should be concurrently sent on another flow to prevent head-of-line blocking.

To realize this behavior, the scheduler assigns the states *disabled* and/or *stale* to each subflow (see Figure 8).

5.1 Implementation

An implementation of the described scheduler is in development. While the crucial part — the detection and recovery from handovers — is not yet ready for comparison general performance results are promising.

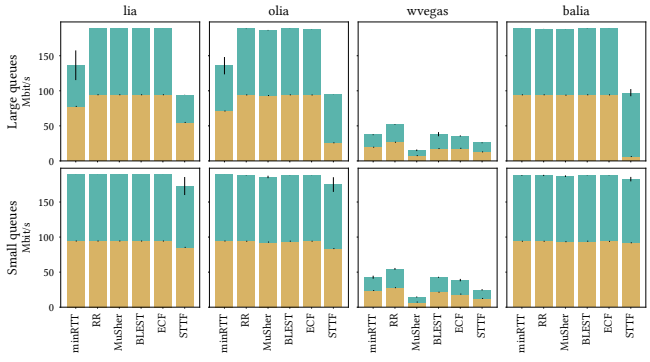

Figure 7: Comparing the impact of the congestion control algorithm on the performance on bandwidth heterogeneous links. Each test was performed with small and with large network queues.

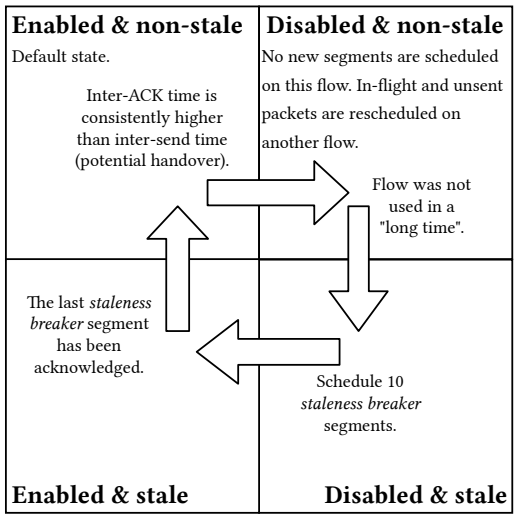

Figure 8: The scheduler marks each flow *disabled* and/or *stale*.

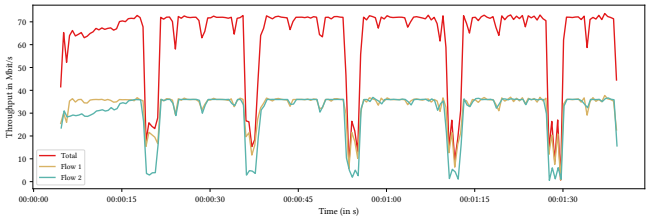

Figure 9: The response of the minRTT scheduler to handover events on one of the two flows in an emulated dual LTE environment. The goodput of each flow and their sum are plotted.

Four comparisons with the minRTT scheduler are presented in Figure 10. Overall, the plots show that our scheduler is not affected by minRTT's undesirable interaction with large network queues.

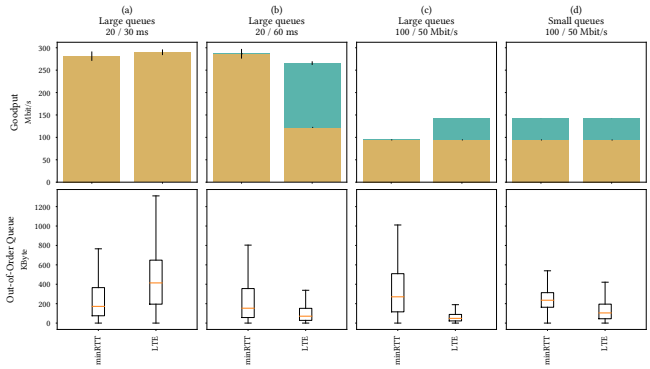

Figure 10: Preliminary performance tests of our LTE scheduler in comparison with the reference minRTT scheduler.

In contrast to minRTT, our scheduler is able to utilize both flows in the large queue scenarios (b) and (c). It does however decide on only using the first flow in scenario (a) where the delay heterogeneity is still smaller than in scenario (b). As the delay-heterogeneity increases, the LTE scheduler carries more data over the impaired flow which is counter intuitive and needs to be examined further. In this scenario, minRTT yields higher total goodput but foregoes an even path utilization.

Plot (d) reveals that our LTE scheduler is on par with minRTT if the bandwidths are heterogeneous and the network queues are small. With the exception of (a), LTE also induces a noticeable lower average out-of-order queue utilization.

6 LESSONS LEARNED

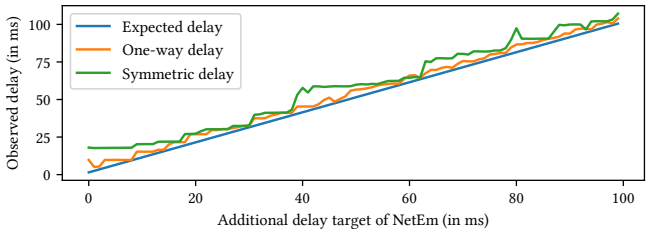

Figure 11: The actually measured delay that is emulated by NetEm. "One-way" delay is only applied to the up-link. "Symmetric" delay means that one half of the expected total delay is applied to each up- and down-link.

We used the NetEm module of Linux' traffic control (*tc*) to emulate network delays, packet loss rates, and more. To limit the available bandwidth of a path, multiple options are available. Either the NetEm parameter `rate` or a second queuing discipline such as *tbf* (Token Bucket Filter) or *htb* (Hierarchy Token Bucket) must be employed. During testing it became apparent that the choice of mechanism has an influence on the observed behavior of MPTCP. The range of configuration options of tbf (in particular *rate*, *burst*, and *latency* / *limit*) and multiple integration options ("Should the packets be first processed by NetEm and then rate-limited or the other way around?") make it complex to understand the impact of

each configuration. The NetEM `rate` option accepts no parameters that were meaningful for our use-case. In the end, we used that to carry out the final tests.

Throughout the process, possible side effects of NetEm had to be taken into account. A documented problem is the interaction of NetEm and TCP Small Queues (TSQ) if NetEm is configured on the sender; in this case, TSQ does not consider packets as "queued" that are currently delayed by NetEm [13, 24]. We have tried to avoid such implementation-specific issues by configuring NetEm on physically-separated middleboxes.

However, neither is this setup foolproof as it can still suffer from problems that impair the quality of the network emulation. As Moe and Jurgelionis et al. have shown, the granularity of the kernel timer greatly influences the accuracy of the delay emulation [15, 24]. To simulate very low delays (e.g., 2ms), High Resolution timer support is necessary as even a 1000 Hz tick rate induces a much higher average additional delay of 5.5ms [24]. While the routers we used run a kernel with support for HRTimers, we still see that the induced delay is always higher than expected (see Figure 11). The problem is exacerbated if it is applied "symmetrically", i.e., to both up- and down-link.

Even though NetEm is a well-known tool that is used by many researchers, care has to be taken when using it to emulate specific network conditions. The described problems lower the trust in the measurement results obtained with emulated network settings. Apart from measuring the actual behavior of NetEm, doing comparison runs over real networks could help sanity checking measurement results.

7 CONCLUSION

Today's devices are often equipped with multiple network interfaces and MPTCP is a promising approach to capitalize on that opportunity. Central to MPTCP's performance is the behavior of the packet scheduler. We have gathered available scheduler implementations, ported them to a common framework base version and setup a controlled testbed to present a fair performance comparison that could previously not have been given.

We find that the size of the intermediate network queues has a significant effect on the schedulers' performance. On the one hand, minRTT struggles to utilize both flows if those queues are large; on the other hand, MuSher shows strikingly low goodput with small queues. Using our testbed, we are able to accurately control this parameter. Its impact on the scheduling performance has not been reported to the best of our knowledge.

The out-of-order queue size has shown to be the parameter next to goodput on which a scheduler's performance has to be measured. Even if two performances are equal in terms of goodput they often show differences concerning the OFO queue size.

REFERENCES

[1] Behnaz Arzani, Alexander Gurney, Shuotian Cheng, Roch Guerin, and Boon Thau Loo. 2014. Impact of path characteristics and scheduling policies on MPTCP performance. In *2014 28th International Conference on Advanced Information Networking and Applications Workshops*. IEEE, 743–748.

[2] Nico Becker, Amr Rizk, and Markus Fidler. 2014. A measurement study on the application-level performance of LTE. In *2014 IFIP Networking Conference*. IEEE, 1–9.

[3] Olivier Bonaventure, Christoph Paasch, and Gregory Detal. 2017. Use Cases and Operational Experience with Multipath TCP. RFC 8041. https://doi.org/10.17487/RFC8041

[4] Olivier Bonaventure and S Seo. 2016. Multipath TCP deployments. *IETF Journal* 12, 2 (2016), 24–27.

[5] Yung-Chih Chen and Don Towsley. 2014. On bufferbloat and delay analysis of multipath TCP in wireless networks. In *2014 IFIP Networking Conference*. IEEE, 1–9.

[6] Christoph Paasch, Sebastien Barre et al. [n.d.]. Multipath TCP in the Linux Kernel. https://multipath-tcp.org

[7] Xavier Corbillon, Ramon Aparicio-Pardo, Nicolas Kuhn, Géraldine Texier, and Gwendal Simon. 2016. Cross-layer scheduler for video streaming over MPTCP. In *Proceedings of the 7th International Conference on Multimedia Systems*. 1–12.

[8] Pingping Dong, Jingyun Xie, Wensheng Tang, Naixue Xiong, Hua Zhong, and Athanasios V Vasilakos. 2019. Performance evaluation of multipath TCP scheduling algorithms. *IEEE Access* 7 (2019), 29818–29825.

[9] Simone Ferlin, Özgü Alay, Olivier Mehani, and Roksana Boreli. 2016. BLEST: Blocking estimation-based MPTCP scheduler for heterogeneous networks. In *2016 IFIP Networking Conference (IFIP Networking) and Workshops*. IEEE, 431–439.

[10] Alan Ford, Costin Raiciu, Mark J. Handley, and Olivier Bonaventure. 2013. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824. https://doi.org/10.17487/RFC6824

[11] Alexander Froemmgen, Jens Heuschkel, and Boris Koldehofe. 2018. Multipath tcp scheduling for thin streams: Active probing and one-way delay-awareness. In *2018 IEEE International Conference on Communications (ICC)*. IEEE, 1–7.

[12] Alexander Froemmgen, Amr Rizk, Tobias Erbshaeusser, Max Weller, Boris Koldehofe, Alejandro Buchmann, and Ralf Steinmetz. 2017. A Programming Model for Application-defined Multipath TCP Scheduling. In *ACM/IFIP/USNIX Middleware*.

[13] Alexander Frömmgen. 2017. Mininet/Netem Emulation Pitfalls: A Multipath TCP Scheduling Experience. *Technical Report* (2017).

[14] Per Hurtig, Karl-Johan Grinnemo, Anna Brunstrom, Simone Ferlin, Özgü Alay, and Nicolas Kuhn. 2018. Low-latency scheduling in MPTCP. *IEEE/ACM Transactions on Networking* 27, 1 (2018), 302–315.

[15] Audrius Jurgelionis, Jukka-Pekka Laulajainen, Matti Hirvonen, and Alf Inge Wang. 2011. An empirical study of netem network emulation functionalities. In *2011 Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 1–6.

[16] Ramin Khalili, Standard Track, Nicolas Gast, and Jean-Yves Le Boudec. 2014. Opportunistic Linked-Increases Congestion Control Algorithm for MPTCP. *Internet Engineering Task Force, draft-khalili-mptcp-congestion-control-05* (2014).

[17] Bruno YL Kimura, Demetrius CSF Lima, and Antonio AF Loureiro. 2017. Alternative scheduling decisions for multipath TCP. *IEEE Communications Letters* 21, 11 (2017), 2412–2415.

[18] Bruno YL Kimura, Demetrius CSF Lima, and Antonio AF Loureiro. 2020. Packet Scheduling in Multipath TCP: Fundamentals, Lessons, and Opportunities. *IEEE Systems Journal* (2020).

[19] Bruno Yuji Lino Kimura and Antonio Alfredo Frederico Loureiro. 2018. MPTCP Linux Kernel Congestion Controls. arXiv:1812.03210 [cs.NI]

[20] Nicolas Kuhn, Emmanuel Lochin, Ahlem Mifdaoui, Golam Sarwar, Olivier Mehani, and Roksana Boreli. 2014. DAPS: Intelligent delay-aware packet scheduling for multipath transport. In *2014 IEEE International Conference on Communications (ICC)*. IEEE, 1222–1227.

[21] Li Li, Ke Xu, Tong Li, Kai Zheng, Chunyi Peng, Dan Wang, Xiangxiang Wang, Meng Shen, and Rashid Mijumbi. 2018. A measurement study on multi-path tcp with multiple cellular carriers on high speed rails. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 161–175.

[22] Yeon-sup Lim, Erich M Nahum, Don Towsley, and Richard J Gibbens. 2017. ECF: An MPTCP path scheduler to manage heterogeneous paths. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. 147–159.

[23] Mat Martineau and Matthieu Baerts. 2019. Mutipath TCP Upstreaming. *Linux Plumbers Conference 2019* (Sep 2019).

[24] Anders G. Moe. 2013. Implementing Rate Control in NetEm. http://home.ifi.uio.no/paalh/students/AndersMoe.pdf

[25] Christoph Paasch and Olivier Bonaventure. 2014. Decoupled from IP, TCP is at last able to support multihomed hosts. *ACM Queue (Print): tomorrow's computing today* 12, 2 (2014).

[26] Christoph Paasch, Simone Ferlin, Ozgu Alay, and Olivier Bonaventure. 2014. Experimental evaluation of multipath TCP schedulers. *Proceedings of the 2014 ACM SIGCOMM workshop on Capacity sharing workshop - CSWS '14* (2014). https://doi.org/10.1145/2630088.2631977

[27] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. 2011. Improving datacenter performance and robustness with multipath TCP. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 266–277.

[28] Costin Raiciu, Mark Handley, and Damon Wischik. 2011. *Coupled congestion control for multipath transport protocols*. Technical Report. IETF RFC 6356, Oct.

[29] Swetank Kumar Saha, Shivang Aggarwal, Rohan Pathak, Dimitrios Koutsonikolas, and Joerg Widmer. 2019. MuSher: An Agile Multipath-TCP Scheduler for Dual-Band 802.11 ad/ac Wireless LANs. In *The 25th Annual International Conference*

on Mobile Computing and Networking. 1–16.

[30] Tanya Shreedhar, Nitinder Mohan, Sanjit K Kaul, and Jussi Kangasharju. 2018. QAware: A cross-layer approach to MPTCP scheduling. In *2018 IFIP Networking Conference (IFIP Networking) and Workshops*. IEEE, 1–9.

[31] Lars Stratmann, Brenton Walker, and Vu Anh Vu. 2020. Realistic Emulation of LTE With MoonGen and DPDK. In *Proceedings of the 14th International Workshop on Wireless Network Testbeds, Experimental evaluation & Characterization*. 87–94.

[32] The Linux Foundation. [n.d.]. TCP Probe. https://wiki.linuxfoundation.org/networking/tcpprobe

[33] Anwar Walid, Qiuyu Peng, Jaehyun Hwang, and S Low. 2016. Balanced linked adaptation congestion control algorithm for MPTCP. *Internet Engineering Task Force, Internet-Draft draft-walid-mptcp-congestion-control-04* (2016).

[34] Jing Wang, Yufan Zheng, Yunzhe Ni, Chenren Xu, Feng Qian, Wangyang Li, Wantong Jiang, Yihua Cheng, Zhuo Cheng, Yuanjie Li, et al. 2019. An active-passive measurement study of tcp performance over lte on high-speed rails. In *The 25th Annual International Conference on Mobile Computing and Networking*. 1–16.

[35] Mingwei Xu, Yu Cao, and Enhuan Dong. 2016. Delay-based Congestion Control for MPTCP. *Internet Engineering Task Force, Internet-Draft draft-xu-mptcp-congestion-control-04* (2016).

[36] Fan Yang, Qi Wang, and Paul D Amer. 2014. Out-of-order transmission for in-order arrival scheduling for multipath TCP. In *2014 28th International Conference on Advanced Information Networking and Applications Workshops*. IEEE, 749–752.