Alexander Frömmgen, Amr Rizk, Tobias Erbshäußer, Max Weller, Boris Koldehofe, Alejandro Buchmann, Ralf Steinmetz
A Programming Model for Application-defined Multipath TCP Scheduling In: Proceedings ACM/IFIP/USENIX Middleware, December 2017

A Programming Model for Application-defined Multipath TCP Scheduling

Alexander Frömmgen Amr Rizk Tobias Erbshäußer Max Weller
Boris Koldehofe Alejandro Buchmann Ralf Steinmetz
TU Darmstadt, Germany
alexander.froemmgen@kom.tu-darmstadt.de

Abstract

Multipath TCP enables remarkable optimizations for throughput, load balancing, and mobility in today's networks. The design space of Multipath TCP scheduling, i.e., the application-aware mapping of packets to paths, is largely unexplored due to its inherent complexity. Evidence in this paper suggests that an application-aware scheduling decision, if leveraged right, pushes Multipath TCP beyond throughput optimization and thereby provides benefits for a wide range of applications.

This paper introduces a high-level programming model that enables application-defined Multipath TCP scheduling. We provide an efficient interpreter and eBPF-based runtime environment for the Linux Kernel, enabling isolated application-defined schedulers in multi-tenancy environments. In combination with a high-level API, our work closes the gap between scheduler specification and deployment. We show the strength of our programming model by implementing seven novel schedulers tackling diverse objectives. Our real world measurements, for example, of an application- and preference-aware scheduler, show that the programming model enables timely scheduling decisions to retain fine-grained throughput objectives. Further measurements of a novel HTTP/2-aware scheduler show significantly improved interactions with upper-layer protocols, e.g., an optimized dependency resolution, while preserving path preferences.

CCS Concepts • **Networks → Transport protocols**; *Network control algorithms*;

Keywords Multipath TCP, Scheduling, Specification Language

1 Introduction

TCP is the network protocol which enables today's computer networks and distributed systems. Multipath TCP (MPTCP) is a recent TCP evolution, that allows splitting a *single* logical transport layer connection over multiple TCP subflows [15]. This increases throughput and reliability while load balancing network resources in data-centers, between data-centers, and in mobile scenarios [8, 42]. At the heart of this logical coupling lies, in addition to congestion control and path management, the *MPTCP scheduling decision*, which is responsible for mapping packets to subflows.

Middleware '17, Las Vegas, NV, USA

DOI: 10.1145/3135974.3135979

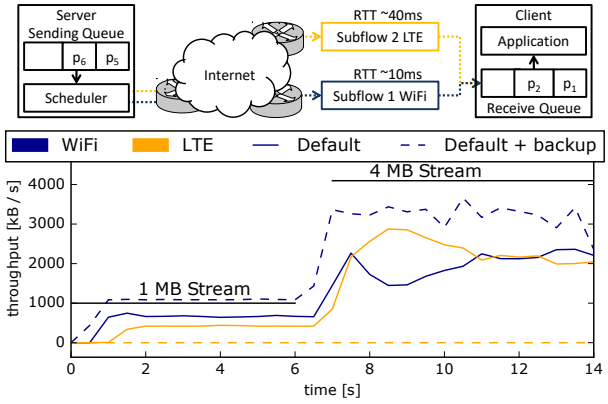


Figure 1. The need for more flexible schedulers: Setup and reproducible measurement result of an interactive streaming session over MPTCP using WiFi and LTE with today's default *MinRTT* scheduler. Neither the default scheduler nor its backup option allow preserving preferences, i.e., exhausting the bandwidth of the faster WiFi subflow before relying on the additional one (LTE) while sustaining the video bitrate of 4MB/s.

Despite the evident benefits of MPTCP, it still struggles with *universal* deployment. In comparison with TCP, Multipath TCP introduces additional complexity as packets of one stream need to be scheduled on multiple subflows before rejoining at the receiving side, as depicted in Fig. 1 (top). This scheduling decision has a substantial impact on the protocol performance [3]. Wrong scheduling decisions may render the advantage of additional paths useless or even reduce the overall performance, e.g., by introducing head-of-line blocking [46].

Today's Multipath TCP implementations are well-tuned for high throughput [42, 44, 47]. Many applications, however, need different scheduling strategies depending on their requirements and characteristics, as well as, the user's context. Applications may have different preferences, e.g., connections may be latency or bandwidth sensitive or devices may be inclined to use one technology (WiFi) over another (cellular). As of today, MPTCP research and implementations lack methods to specify and deploy optimized application- and preference-aware schedulers.

To illustrate the current inability of applications to leverage MPTCP scheduling, Fig. 1 depicts an example of an interactive streaming session running "in the wild" between a cloud provider and a mobile device using off-the-shelf MPTCP [37] with two subflows (WiFi and LTE). The first 6 seconds of the stream are encoded with 1MB/s, the remaining part with 4MB/s. Although the 1MB/s stream is sustainable on the 10ms round-trip time (RTT) WiFi-subflow, we observe that today's default scheduler, denoted

Alexander Frömmgen, Amr Rizk, Tobias Erbshäußer, Max Weller, Boris Koldehofe, Alejandro Buchmann, Ralf Steinmetz
A Programming Model for Application-defined Multipath TCP Scheduling In: Proceedings ACM/IFIP/USENIX Middleware, December 2017

as *MinRTT*, places 30% of the traffic on the higher RTT subflow (LTE, 40ms). This is the result of the throughput and load balancing optimization of the default *MinRTT* scheduler in conjunction with specific timings of the congestion control and TCP small queue (TSQ) optimization. We note that setting the LTE subflow to backup mode does not provide remedy as it practically deactivates the subflow. This becomes evident when the stream quality rises to 4MB/s, where the backup mode does not provide enough throughput. The demand for fine-grained scheduling specifications increases with the availability of more subflows, e.g., for connections between data-centers. This symptomatic example reveals aspects that lie at the heart of this work: MPTCP requires more intelligent schedulers that can be *tailored to the needs and preferences of diverse applications through processing timely subflow information.*

In this paper, we aim to facilitate the development of general purpose and application-tailored schedulers, e.g., for streaming, web, and real-time applications. The development of MPTCP schedulers is very challenging due to the wide span of input variables, such as path latencies, bandwidth estimates or packet drops. Furthermore, MPTCP scheduling is part of the transport protocol implementation in the Linux Kernel to ensure timely scheduling decisions. This makes naive approaches for application-defined schedulers, such as Kernel modules and per-scheduler modifications in the Kernel, infeasible for scalable cloud infrastructure. Additionally, the implementation complexity in the Linux Kernel holds back the implementation and evaluation of novel schedulers. Recent works propose scheduling optimizations for multipath protocols, leaving an implementation and evaluation within the MPTCP Linux Kernel open [10, 19, 31, 54] or avoiding complex changes in existing, underlying schedulers [22].

In essence, our contribution is a high-level programming model for MPTCP scheduling, providing the necessary abstractions for a wide range of scheduler innovations. Programming abstractions enabled networking innovations in recent years, e.g., approaches such as OpenFlow [33] and Click [30] took the community by storm and expanded to network-wide programmability [16] and programming models for network services [2]. We derived the primitives of our programming model based on existing schedulers and our experiences while optimizing schedulers. In addition, we introduce a simple yet powerful scheduling API that enables application-aware scheduling. We further provide an efficient runtime environment in the MPTCP Linux Kernel, closing the gap between the scheduler specification and its execution.[1]

To highlight the potential of application-aware scheduling, we present an HTTP/2-aware scheduler that significantly improves the initial dependency resolution by selectively prioritizing packets while preserving subflow preferences. We further present a wide range of novel schedulers, e.g., *(i)* to boost short flows, *(ii)* to balance the level of redundancy to reduce latency, or *(iii)* to retain target subflow properties in changing environments.

In summary, we contribute:

1. A high-level programming model and API for the specification of Multipath TCP schedulers which abstracts over implementation details.
2. A runtime environment for the scheduler programming model in the Linux Kernel, providing efficient interpreter and eBPF-based evaluation while preserving isolation between applications.
3. A wide range of novel schedulers, showing substantial performance benefits and the potential of our programming model, i.e., for application- and preference-aware MPTCP schedulers.

This paper is structured as follows. Based on a discussion of related work (§2), we introduce our scheduler programming model (§3). We present implementation details of our runtime environment (§4) before providing a systematic derivation of new optimized schedulers (§5). Finally, we discuss the usability and limitations of our programming model (§6) before providing conclusions (§7).

2 Background and Related Work

Packet scheduling is a basic operation in communication networks. From a programming model perspective, recent works [48, 49] raise the abstraction layer for *scheduling in switches* proposing programmable packet scheduling. On end-hosts, configurable traffic shapers already provide abstractions for scheduling decisions *between independent flows*, e.g., based on QoS flags. On the transport layer, the well-known Nagle algorithm and the tail loss probe [14, 45] represent scheduling directives in the TCP stack that reduce packet header overhead and improve loss recovery for short flows, respectively. In addition, abstraction models, e.g., *Readable TCP in the Prolac Protocol Language* [29], were suggested for the development and implementation of *comprehensive transport protocols*, i.e., packet header handling and semantics. In contrast, we focus here on abstractions and execution environments for MPTCP scheduling. Next, we present the MPTCP building blocks and recent works on MPTCP scheduling.

2.1 Multipath TCP – The Building Blocks

Multipath TCP provides the standard socket interface dispensing with the need for application modifications. A logical MPTCP connection is split into subflows, where each subflow behaves like a traditional TCP connection making MPTCP deployable on the Internet [50]. Global sequence numbers are transferred as TCP options to ensure in-order packet delivery.

As of today, the Linux Kernel MPTCP implementation [44] is one of the most widely used MPTCP implementations beside Apple's implementation for the cloud-based assistant system *Siri*. The main building blocks of the Linux implementation are *(i)* the meta socket, *(ii)* the path-manager, *(iii)* the congestion control, and *(iv)* the scheduler. The meta socket is the central abstraction of each MPTCP connection. The path manager decides on the creation and removal of subflows. Compared to the scheduling decision, the path manager has relaxed time constraints, as the creation of sublows requires multiple round-trip times. This was used by recent papers to extend the socket API and enable path manager logic in the userspace [24, 25]. The MPTCP congestion control received much attention, with many proposed algorithms aiming, e.g., at TCP friendliness on shared bottleneck links [28, 43, 53]. Finally, the scheduler decides on which subflow a packet is transmitted. While MPTCP details, such as the global sequence numbers and the used TCP options are specified [15], scheduling is not standardized.

All building blocks are closely tied, e.g., today's schedulers require subflows to be initiated by the path-manager and consider the congestion window maintained by the congestion control block.

[1] Our runtime environment and a detailed tutorial are available at https://progmp.net

Alexander Frömmgen, Amr Rizk, Tobias Erbshäußer, Max Weller, Boris Koldehofe, Alejandro Buchmann, Ralf Steinmetz
A Programming Model for Application-defined Multipath TCP Scheduling In: Proceedings ACM/IFIP/USENIX Middleware, December 2017

Optimizing MPTCP for high throughput effectively lets the congestion control *schedule* the traffic [53], as the scheduler is blocked by the congestion control, which was recently exploited in [41]. For thin and application limited flows, such as *request-response* patterns, congestion windows are not permanently exhausted, reducing the impact of the congestion control on the scheduling decision. Although the achieved application performance, e.g., with respect to latency, depends on all building blocks, we particularly find that research on executable, application-defined MPTCP schedulers has been lacking fundamental abstractions.

2.2 Multipath TCP – Scheduling

Flexible Schedulers Inspired by TCP's pluggable congestion control logic, Paasch et al. [39] presented a pluggable scheduler model for the Linux Kernel. Nikravesh et al. [35] proposed to move the scheduling decision to the userspace – as part of a sophisticated proxy infrastructure – and to provide policies to fix schedulers to applications. In [35], the authors do not discuss details of their calling model, the induced overhead at the sending server, and the impact on the scheduling timeliness. Both works [35, 39] increase flexibility with respect to the scheduler selection, but lack programming abstractions which support application-defined schedulers and provide isolation for cloud environments.

Availability in the Linux Kernel The current MPTCP Linux Kernel implementation is based on the pluggable scheduler model from [39] and comprises the *round robin*, the *default*, and the *redundant* scheduler. The rather simple *round robin* scheduler is known to perform poorly for heterogeneous paths, as slow subflows constrain the overall performance. The *default* scheduler considers the round-trip time (RTT) and the congestion window for each subflow. The scheduler assigns packets to the subflow with the lowest RTT that has not exhausted its congestion window yet [44].[2] The current *redundant* scheduler in the Linux Kernel is a combination of the works in [17, 32] aiming to reduce latency for applications with moderate bandwidth requirements. Although assessing code complexity is difficult, we note that, e.g., the *naive* round robin scheduler already requires 301 lines of code (LOC). Further commits (two critical and six minor fixes) highlight the error susceptibility of today's development methodology. We revisit these schedulers in (§3.4) to discuss semantic details and illustrate the strength of specifying schedulers using our programming model.

Compensate Loss in Short Data-center Flows The works in [7, 27] show how to compensate packet losses in data-center environments using MPTCP to improve the tail flow completion time. The authors of [27] proposed a *Fast Coupled Retransmission* for unacknowledged packets on non-congested paths when a duplicate acknowledgement is received. In [7], packets are recovered on alternative subflows by retransmitting the oldest, unacknowledged packet of the subflow with the highest loss rate when loss is *suspected*. Both independent works use slight variations of the same idea. An analysis of the design decisions, such as the choice of the retransmitted packet, was not in the scope of these approaches.

Video Streaming The authors of [19] proposed an energy-efficient HTTP-based adaptive video streaming over heterogeneous wireless networks. The authors aim to benefit from MPTCP, yet they use traditional TCP for their evaluation. The recent work in [22] presented a preference-aware *Dynamic Adaptive Streaming over HTTP* (DASH) framework denoted as *MP-DASH*. The authors introduce a control loop on top of the scheduler, which controls the *visibility* of subflows to the *default* scheduler. The work in [22] provided an important step towards preference-aware scheduling, however, confined to the chunk-based DASH video streaming application. In contrast, we present a programming model for MPTCP scheduling with fundamental abstractions that enable optimizing various performance metrics for a *wide range of applications* including DASH video streaming. We provide corresponding evaluations in (§5).

Discussion The wide range of MPTCP scheduling designs and optimizations confirms our observation that there is no *one size fits all* scheduler. We note that existing scheduler optimizations include many design decisions, which are rarely evaluated, partially due to the lack of a deployable implementation. In general, we find that a systematic design of application- and preference-aware schedulers is missing and that it is very hard, if not impossible, to assess the real-world performance of many of the proposed MPTCP schedulers. In particular, a substantial gap between scheduler specifications and the corresponding execution is apparent. This work closes these gaps with a bespoke MPTCP scheduler programming model accompanied by a runtime environment fostering the specifications and deployment of novel schedulers.

3 Programming Model

This section introduces one of the cornerstone ideas of this work, the MPTCP scheduler programming model which abstracts over implementation details. First, we show how to capture the scheduling environment in a concise model (§3.1) and discuss an extended API that enables application-aware schedulers (§3.2). Then, we use this model to specify schedulers (§3.3) before demonstrating its concepts and expressiveness by providing examples of existing mainline schedulers (§3.4).

3.1 Model of the MPTCP Scheduling Environment

Fig. 2 shows an overview of the Multipath TCP scheduling environment. The scheduler connects two building blocks: *(i)* the sending queues and *(ii)* the subflows. The application uses the TCP socket to push new data packets p_i from the userland to the sending queue $Q = [p_1, \ldots, p_m]$ in the Kernel. The scheduler pushes the packets from Q into subflow queues such as Q_{sbf1} for subflow #1. For each packet, the scheduler chooses at least one subflow out of the set of available subflows. Note that packets might be mapped to multiple subflows for redundant transmission. Packets pushed from the sending queue Q are stored in QU, which is for *unacknowledged* packets in flight. In case a subflow suspects a packet drop, e.g., due to three duplicate acknowledgements or a retransmission timeout, this packet is automatically added to the *reinjection queue* RQ. Note that packets

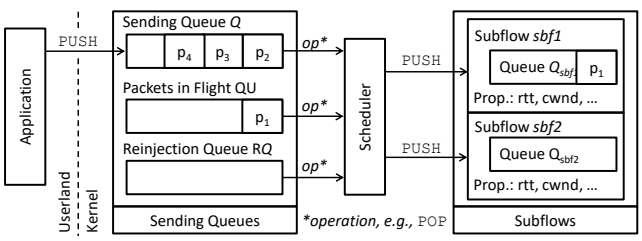


Figure 2. Model of the scheduler environment.

[2]Related work rarely mentions that the default scheduler only considers subflows which are not throttled due to *TCP small queue* condition and are not in a loss state.

are not reinjected into the sending queue Q. Acknowledged packets are automatically removed from *all* queues. The MPTCP receiver ensures in-order delivery to the receiver application regardless of the used scheduler at the sending side (§4).

3.2 API for Application-defined MPTCP Scheduling

In the following, we discuss requirements for an extended *scheduling API*. Based on this, we present a corresponding scheduling API as part of our programming model which significantly expands the design space of MPTCP schedulers.

API Requirements The scheduling API has to allow the application to interact with the scheduler and to provide all important information. Consider, for example, a database where *small requests*, which usually consist of a few packets, may significantly benefit from redundancy while introducing a limited overhead. In contrast, *heavy* database *responses* can be transmitted throughput-optimized on the same connection. This example motivates the need for a per-connection scheduler choice and means for the application to inform the scheduler about changing intents. A scheduling intent is a piece of information provided by the application to the scheduler to request a specific scheduling behavior. Note that for a differentiated behavior on packet granularity, e.g., pushing latency sensitive packets, we require the application to be able to provide per-packet scheduling intents.

In addition to the traditional TCP socket interface for the support of legacy applications, the extended API includes the following primitives for the interaction of the application with the scheduler.

Choosing a Scheduler The extended scheduler API supports a per-MPTCP-connection scheduler choice permitting a concurrent use of different schedulers for different connections. The application is allowed to *load its own, application-defined* schedulers or reuse loaded schedulers to reduce compilation overhead.

Switching schedulers at runtime is disadvised, as we experienced that this provokes inconsistent states, e.g., due to different assumptions and strategies of schedulers. Instead, we encourage to rely on setting registers, as presented in the following.

Setting Registers An application can *set registers* in the scheduler which enables, for example, different scheduling modes. Such modes comprise, e.g., an aggressive retransmission mode for latency sensitive content, or a *data flushing* mode that shortens transmission time when the application has no more data to send.

Packet Properties An application can set packet properties for differentiated handling of packets. High priority packets may, for example, be transferred redundantly or additionally sent on backup subflows. Further, packets can be forced on certain subflows depending on their properties, e.g., to ensure privacy.

3.3 Specifying Schedulers

In the following, we discuss requirements and targets for the programming model. Based on this, we present the according programming operations and scheduler triggering events.

Requirements and Targets The programming model has to enable a timely and efficient execution of application-defined schedulers in the range of a few hundred nano seconds, as shown in §4.1. This requires a precise specification of the scheduling logic, which is preferably automatically optimizable by the runtime environment. Time consuming operations, such as dynamic memory allocations, must not be necessary.

In addition to the execution efficiency, the programming model should enable crisp expressions of the scheduling logic avoiding both implementation errors and ambiguity. Typical pitfalls in the selection of both packets and subflows relate to the establishment of new subflows and the sudden failure of established subflows. In a pure C implementation, the disappearance of a subflow might easily lead to stale references and crashes of the operating system. Ideally, accessing stale subflow references should not only be prohibited or fail gracefully, but should be impossible by design. Further, handling references to packets, for example, for the assignment to multiple subflows, is a substantial source for errors. In particular, packets must not be lost, e.g., when the target subflow ceases to exist after the packet is already removed from the sending queue.

Programming Operations The programming model provides the environment model of §3.1 as first class citizen, i.e., a set of subflows denoted as `SUBFLOWS` and the queues `Q`, `QU` and `RQ` with packet entities. Accordingly, `TOP` and `POP` are the basic operations for selecting and removing packets from queues, respectively. The `PUSH` operation schedules a packet on a subflow.

For both the subflow and the packet selection, we rely on a declarative specification with `FILTER(`*element* `=>` *boolean_predicate(element)* `)` and `MIN(`*element* `=>` *integer_ predicate(element)* `)` operations. The declarative subflow choice may use various subflow properties such as (*i*) the round-trip time (RTT), (*ii*) the congestion window (cwnd) as maintained by the congestion control algorithm, and (*iii*) the number of packets in flight. For example, `SUBFLOWS.FILTER(sbf => sbf.RTT_AVG < 10).MIN(sbf => sbf.RTT_VAR)` selects the subflow with the minimal RTT variance out of all subflows which have an average RTT less than 10ms. Similarly, queues are filtered based on packet properties, e.g., the packet size or whether and when the packet was sent on a certain subflow. The model provides additional functions, e.g., to check whether a subflows receive window can accommodate new data using `HAS_WINDOW_FOR(<packet>)`.

The declarative specification based on packet queues and the set of available subflows avoids complex control flows as well as error-prone pointer references and index operations. Furthermore, the programming model only allows side effects as part of `PUSH` operations. This avoids common pitfalls, e.g., by prohibiting side effects in *boolean_predicate*s, we ensure that packets are not accidentally removed due to a `Q.POP().RTT` expression in a condition.

In addition to the declarative elements, the programming model provides simple control flow operations, such as `IF .. ELSE ...` Essentially, these operations simplify the development of different scheduling modes. Furthermore, the programming model supports variables in a single-assignment form. The variable handling avoids dynamic type errors by design, as each variable has the implicit type of its initial assignment. Variables, subflow and packet properties are immutable during a single scheduler execution. This simplifies the scheduler specification, as it provides the illusion of a single-threaded program for the developer. Additionally, immutable values enable efficient control flow optimizations, as discussed in (§4.1).

For performance and scalability reasons, i.e., to avoid dynamic memory allocation during scheduler executions, each scheduler keeps a limited state of integer values, denoted as `Registers`, that are used to provide application-awareness (§3.2) or to retain state between scheduler executions, e.g., for round-robin schedulers (§3.4). Preventing additional state transfer between scheduler executions, i.e., of references to subflows, avoids stale references by design.

Alexander Frömmgen, Amr Rizk, Tobias Erbshäußer, Max Weller, Boris Koldehofe, Alejandro Buchmann, Ralf Steinmetz
A Programming Model for Application-defined Multipath TCP Scheduling In: Proceedings ACM/IFIP/USENIX Middleware, December 2017

Table 1. Key concepts of the programming model.

	Concept
Subflow properties	e.g., `RTT`, `RTT_VAR`, `CWND`, `SKBS_IN_FLIGHT`, `QUEUED`, `IS_BACKUP`
Packet properties	e.g., `SIZE`, `SENT_ON(sbf)`
Subflow selection	Declarative based on `FILTER`, `MIN` and `MAX` with packet and subflow properties
Packet selection	Declarative based on `FILTER`, `MIN` and `MAX` with packet and subflow properties
Variables	Single assignment, implicit typing
Types	Static type system (`int`, `bool`, `packet`, `subflow`, `subflow list`, `packet queue`)
Concurrency	Immutable properties and variables
Side Effects	Restricted to `PUSH` operations
Exceptions	No exceptions by design

Table 1 provides a summary of these key concepts. Throughout the remainder of this section we use exemplary specifications of intuitive schedulers to show the comprehensiveness of our programming model. Fig. 3 shows an example specification excerpt of a scheduler that uses the subflow with the minimum RTT to `PUSH` the first packet from `Q` if the sending queue `Q` is not empty and there exists at least one subflow. The `POP` operation removes the first packet from `Q`.

```
1  IF (!Q.EMPTY AND !SUBFLOWS.EMPTY) {
2     SUBFLOWS.MIN(sbf => sbf.RTT).PUSH(Q.POP()); }
```

Figure 3. Excerpt of a scheduler specification that pushes packets on the subflow with minimum RTT.

Scheduler Triggering and Execution The *per packet* scheduling decision should be based on the freshest available information and should be taken timely before the packet is actually pushed on the wire. Thus, *scheduling packets in userspace at the moment the data is pushed by the application is not sufficient.* For example, the scheduler would fail to take a reasonable scheduling decision in case all subflows are congestion limited at the time the application pushes data. In this work, we extend the implicit model of Barré et al. [4] and trigger the scheduler execution by a number of events that include the arrival of new packets in `Q` or receiving acknowledgements (Fig. 4). To simplify the scheduler development, the scheduler is not required to handle all packets in `Q` upon one execution. Hence, a scheduler execution rather focuses on a single or a few packets. We discuss the performance impacts of this design decision in (§4).

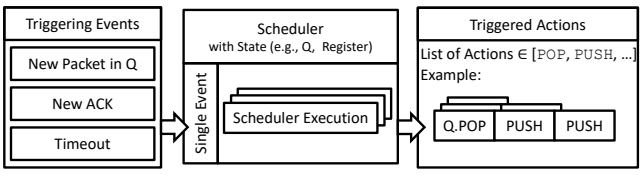


Figure 4. Scheduler calling model.

3.4 Revisiting Existing Schedulers

Here, we show the expressiveness of our programming model by revising and tuning notable MPTCP schedulers from the related work. Even though we possess compact comprehensive scheduler specifications, we only show excerpts that reflect the *key functionality* of these schedulers and the programming model.

Round Robin Scheduler Fig. 5 presents a round robin scheduler, which implements a cyclic subflow index in register `R1` and skips subflows with an exhausted congestion window for work conserviness. Configuration parameters of the existing round robin implementation in the Linux Kernel result in minor modifications.

```
1  VAR sbfs = SUBFLOWS.FILTER(sbf =>
2        !sbf.TSQ_THROTTLED AND !sbf.LOSSY);
3  IF (R1 >= sbfs.COUNT) { SET(R1, 0); }
4  IF (!Q.EMPTY) {
5     VAR sbf = sbfs.GET(R1);
6     IF (sbf.CWND > sbf.SKBS_IN_FLIGHT + sbf.QUEUED) {
7        sbf.PUSH(Q.POP()); }
8     SET(R1, R1 + 1); }
```

Figure 5. Excerpt of the round-robin scheduler.

(Default) Minimum RTT Scheduler The current default scheduler in the MPTCP Linux Kernel [44] pushes packets from the sending queue on the subflow with the lowest round-trip time and unexhausted congestion window. Basically, replacing the highlighted parts in Fig. 5 with `sbfs.FILTER(sbf => sbf.CWND > sbf.QUEUED + sbf.SKBS_IN_FLIGHT).MIN(sbf => sbf.RTT);` leads to the default scheduler. Our programming model allows to easily explore a wide range of related schedulers, e.g., considering the RTT variance for jitter reduction, or the RTT ratio of the different subflows to minimize packet dispersion.

Backup Semantics The default scheduler only uses backup subflows `IF(SUBFLOWS.FILTER(sbf => !sbf.IS_BACKUP).EMPTY)`. Our programming model provides the flexibility to consider additional metrics, such as the round-trip times or capacities, for the backup decision (leveraged in (§5.4)).

Opportunistic Retransmission Feature The *opportunistic retransmission* [44] feature extends the default scheduler by retransmitting packets from a slower subflow (higher RTT) on a faster subflow (lower RTT) when the receive window is blocked, i.e., using `IF(!minRttSbf. HAS_WINDOW_FOR(Q.TOP))`. Our programming model allows to explore the wide range of related schedulers, e.g., to proactively detect such a situation and retransmit packets before the receive window blocks or to selectively retransmit these packets on multiple subflows.

Redundant Scheduler The works in [17, 32] define schedulers that redundantly send packets on all available subflows. As the first received copy is used, this approach trades throughput for latency. This scheduler is particularly interesting, as the term redundant here is *ambiguous*. It does not specify, for example, the behavior given dynamic subflow conditions, e.g., as new subflows become available. Our programming model eliminates this ambiguity by providing an *explicit and comprehensive specification*, e.g., pushing a packet `VAR skb = Q.POP()` on all subflows `FOREACH(VAR sbf IN SUBFLOWS){sbf.PUSH(skb);}`. We leverage our programming model in (§ 5.1) to specify flavors of MPTCP redundancy for fine-grained performance control.

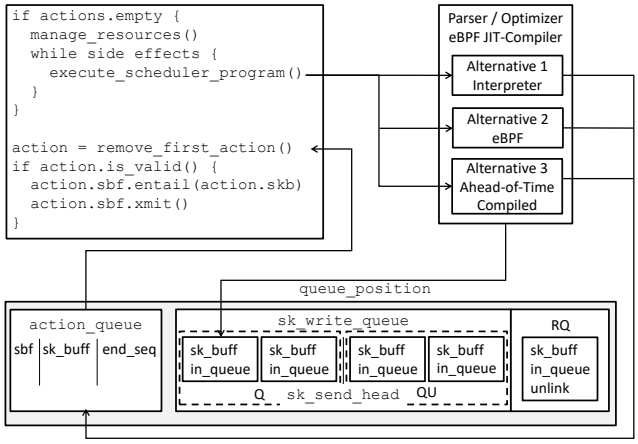


Figure 6. Implementation overview, showing the general control flow of the calling model (top left), three alternative execution environments (top right), and the supporting data-structures (bottom).

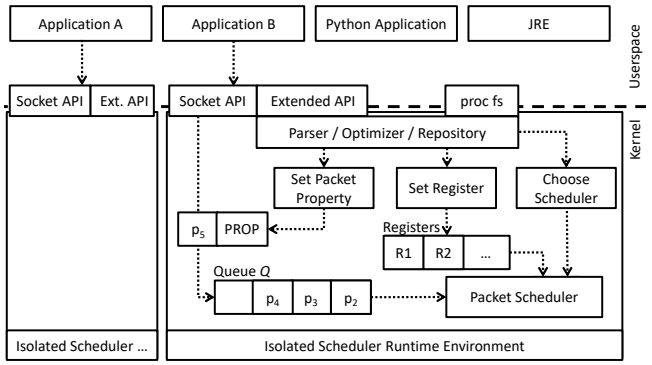


Figure 7. Extended scheduler API implementation.

4 Implementation

In this section, we provide implementation details of the scheduler runtime environment and the packet handling at the receiver. Further, we discuss the computational overhead of our runtime environment and show that it retains scheduling throughput performance. Finally, we provide a discussion of targeted developers, usability and limitations of the programming model.

4.1 Runtime Environment

Scheduler Location and Calling Model While designing the implementation of the calling model (§3.3) we considered two alternatives: *i)* Userspace up-call and *ii)* in-Kernel processing.[3] While the userpace up-call simplifies the development effort for the runtime environment, it introduces latency and computational overhead. We used a *netlink*-based prototype on the same infrastructure used in (§4.3), where we observed that a single up-call requires about 2.4 μs. A single scheduler execution in the Linux Kernel implementation requires, however, about 0.2 μs. Although optimizations for Kernel-userspace communication, e.g., batch processing in the OpenVSwitch [40], reduce this overhead, we implement the runtime environment in the Kernel to provide maximum performance.

Implementation Details We implemented the scheduler runtime environment consisting of nearly 20,000 LOC in the Linux Kernel (version 4.1.20) and integrated it in the existing MPTCP implementation (version 0.90). Fig. 6 illustrates the overall implementation. The runtime manages an own `queue_position` pointer in the `sk_write_queue` in addition to the established `sk_send_head` pointer. In conjunction with additional flags in the `sk_buff` packets, e.g., the `in_queue` flag, this provides an augmented queue on top of the existing queues. This augmentation implements the abstractions of the programming model, e.g., to enable `POP` operations in the middle of the queue or push `TOP` packets without removing them from the sending queue `Q`. The scheduler execution and the actual `PUSH` operations are internally decoupled with an `action_queue` to speed-up the evaluation while preserving visible side-effects of the programming model. The implementation further consists of three alternative execution environments: *(i)* an *interpreter*, *(ii)* an *ahead-of-time compiler*, and *(iii)* an *eBPF-based just-in-time compiler*. Alternative 1, the interpreter, provides the baseline implementation. The interpreter does not require changing executable machine instructions at runtime, and works without additional support of the operating system. Alternative 2, the ahead-of-time compiler, generates and compiles C functions to be called at runtime. This allows the programming model to be executed without the need of a parser or interpreter in the Kernel. Alternative 3, the eBPF-based Just-In-Time compiler, translates the intermediate representation into the machine independent eBPF assembly language and uses the eBPF Kernel infrastructure [9] to compile to native code at runtime. We provide more details about the eBPF-based implementation later in this section.

API Implementation and Toolchain Fig. 7 shows the embedding of the runtime environment and the presented application interface. We use *sockopts* for the extended API. The runtime additionally provides an extensive `proc`-based interface with debugging and performance statistics, e.g., performance profiling traces based on the control flow representation of the scheduler specification.

Furthermore, we implemented a Python library for userspace applications. As shown in the example in Figure 8, this library hides the implementation complexity of the underlying socket API and the Linux Kernel. After the usual connection establishment (lines 2–3), the application developer can load own scheduler specifications (`schedProgStr`, line 5), choose schedulers (`setScheduler`, line 9) and set registers (`setRegister`, line 12) naturally in Python.

```
 1 from progmp import ProgMP
 2 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
 3 s.connect(targetIp, targetPort)
 4 try:
 5    ProgMP.loadScheduler(schedProgStr, schedulerName)
 6 except:
 7    print "Scheduler loading error…"
 …
 8 try:
 9    ProgMP.setScheduler(s, schedulerName)
10 except:
11    print "Setting scheduler failed…"
 …
12 ProgMP.setRegister(s, ProgMP.R1(), 5)
```

Figure 8. Usage example in Python: The Python library hides all socket API and Kernel details.

[3]Note that the programming model abstracts over implementation details, which allows the scheduler developer to be agnostic with respect to both alternatives.

Runtime Optimizations We implemented a wide range of optimizations based on the declarative language elements. `FILTER`s, for example, are evaluated using late materialization. Two additional optimizations are particularly interesting: *i)* constant subflow number and *ii)* compressed executions. As the number of MPTCP subflows changes rarely, the JIT-compiler optimizes for a constant number of subflows and returns to the original version otherwise. The second optimization increases the number of actions per scheduler execution to compress the number of executions. For example, to push data from `Q` on the subflow that has the lowest RTT *and* that has not exhausted its congestion window, we do the following: Instead of executing the term `SUBFLOWS.FILTER(sbf => sbf.SKBS_IN_FLIGHT < sbf.CWND).MIN(sbf => sbf.RTT).PUSH(Q.POP())` multiple times, the program is transformed to recurrently execute the `PUSH`-operation on the resulting `MIN` subflow (i.e, the subflow with lowest RTT) until its congestion window is full. Note that all optimizations are enabled by the abstractions of the programming model.

eBPF Compilation To further speed up the scheduler execution, we decided to use *eBPF*, which originated in the need to push small programs, e.g., filter for network traffic, into the Linux Kernel. Thus, eBPF is a recent approach in the Linux Kernel to facilitate the safe execution of dynamically loaded programs in the Kernel [9]. The eBPF infrastructure consists of an own special-purpose virtual machine for the platform independent eBPF assembly language in the Linux Kernel. Additionally, the eBPF Kernel infrastructure provides support to compile eBPF assembly to native code at runtime. We found the existing userspace eBPF infrastructure, which compiles C to eBPF assembly, insufficient, as we required to compile from the scheduler intermediate representation to eBPF assembly *in the Kernel*. To this end, we implemented an own cross-compiler in the Kernel. This cross compiler uses an extended version of the linear scan register allocation, specifically, the *Second-Chance Binpacking* algorithm [51], which is computationally superior to iterative computations arising in graph-coloring register allocation.

Based on our cross-compiler, the actual eBPF compilation replaces most interpreter calls with native code and combines scheduler primitives, such as `FILTER`, reducing the number of loops and function calls. The compilation is executed concurrently in a separate thread, therefore not harming network performance.

4.2 Receiver-Side Packet Handling

At the receiving side, all in-order packets should be pushed to the application as soon as possible. The sequence number mapping of MPTCP, as specified in the RFC [15], provides the necessary information to map subflow sequence numbers to meta sequence numbers given that each packet carries a full mapping that contains the packet itself. We observed, however, that the current implementation in the Linux Kernel *does not deliver* all packets as soon as possible. For certain packet loss and out-of-order patterns between subflows, in-order data is not pushed to the application. This is due to the multilayer queue architecture with both a *receive* and an *out-of-order* queue per subflow and meta socket. Only *in-subflow-order* packets without gaps are pushed from the subflow to the meta socket, even though other packets might also fit in-order in the meta receive queue.

We implemented all necessary changes at the receiver-side to ensure fastest possible packet handling. This is in particular difficult, as the packets in the subflow queues are sorted by subflow sequence number, but have to be *read* in order of the data sequence number. We appreciated the use of `packetdrill` [6], a tool that uses crafted input packet traces for testing the Linux network stack, to extensively test the receiver side packet handling for incoming packet combinations. During the evaluation, we noticed that the optimizations are particularly important for sophisticated schedulers, and are rarely required for the established ones.

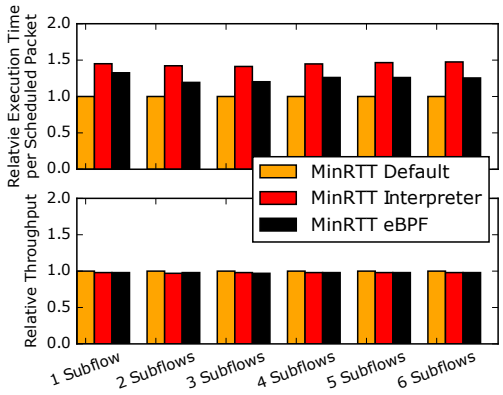


Figure 9. Overhead evaluation of our implemented runtime environment for the programming model.

4.3 Overhead

We optimized our runtime environment implementation, as overhead is a main consideration regarding abstractions. We empirically evaluated our implementation using two bare-metal servers with 64 cores and 128 GB of RAM connected on a 10Gb/s network.

Throughput and Computational Overhead An isolated measurement of the schedulers' execution times in the Linux Kernel is challenging, as collecting and storing measurements in the range of nanoseconds introduces significant overhead. We compare the execution times of the C-based default scheduler implementation with a semantically equivalent scheduler specified in our programming model. Fig. 9 (bottom) shows the maximum throughput for a CPU-limited connection. *The total throughput remains unchanged throughout all schedulers*, showing that the scheduler overhead does not affect the achievable throughput.

In terms of per packet average execution times, we observe that the eBPF-based optimization reduces the relative execution time of the interpreter from ~ 144% to ~125%. The impact of the number of subflows is marginal. In additional measurements, we found that the execution times and the effectiveness of the implemented optimizations depend on a multitude of factors, such as the network characteristics and the used scheduler operations.

The increased execution time and potential consequences for higher throughput networks present no limitations for two reasons: *i)* the total execution time is in the range of nanoseconds, which is magnitudes below most network link latencies and *ii)* for purely throughput-optimized applications, the application developer can still rely on today's default scheduler. Our proposed programming model, however, is in particular useful for optimizations *beyond* throughput, as shown in §5.

Number of Schedulers As previously loaded schedulers can be reused, we anticipate a limited number of different schedulers.

Table 2. Unexplored design space for MPTCP schedulers.

Category	Goal	Signaled Appl. Info.	Pref.	Approach / Design Space	(§)
Probing	Timely RTT/capacity estimates			Probe subflows of interest	
Redundancy	Minimize latency vs.			Prefer new or old packets?	(§5.1)
	induced overhead			Partial redundancy	
	(e.g., flow completion time in lossy networks)		√	Use backups for redundancy depending on non-backup RTT average & variance, loss,...	
Handover	Smooth WiFi/LTE handover			Retransmit on new subflow	(§5.2)
Heterogeneous	Minimize FCT	Flow size		Optimal schedule	
Subflows		signaled		Avoid slow subflow at end of flow	
		End of flow		Compensation of sched. decisions	(§5.3)
		signaled		Selective compensation	(§5.3)
Preference	Ensure RTT	Tolerable RTT	√	Use backups depending on non-backup RTT value, difference or variance above threshold?	(§5.4)
	Ensure throughput	Min. throughput	√	Use backups if throughput insufficient	(§5.4)
	Ensure deadline	Target deadline	√	Use backups if deadline would be violated	
Higher	TLS-aware	TLS records		Coherence of TLS records	
Protocols	HTTP/2-aware	Logical chunk	√	Content-aware strategies	(§5.5)

The required memory depends on the concrete scheduler, e.g., the round robin scheduler requires 3048 Byte. Each instantiation of the scheduler requires additional 328 Byte. Compared to the overall network stack, the memory overhead of our runtime environment does not restrict the adoption.

5 Evaluation

In this section, we evaluate the benefits of our programming model by systematically specifying novel application- and preference-aware schedulers. Both the specification of novel schedulers, as well as the measurement of the achievable improvements are fundamental contributions of this paper.

Table 2 provides a non-comprehensive categorization of the vast unexplored MPTCP scheduler design space. This covers features which improve existing schedulers, such as a recurrent probing of unused subflows. As thin flows typically do not use all subflows, fresh RTT estimates significantly improve the scheduling decision in dynamic environments. Design decisions for redundant transmission include the choice of new and old packets (§5.1), the number of duplicates, and the incorporation of backup subflows. Table 2 further shows examples of using application information to improve the flow completion time in heterogeneous environments (§5.3). In particular, the combination of application information and preferences enables significant improvements to ensure throughput while minimizing the usage of costly subflows (§5.4). Finally, we show the strong benefits of differentiated scheduling during a single connection. Here, we present a case study of HTTP/2-aware scheduling that demonstrates how all building blocks of the programming model interact to improve page load times and achieve a reduced usage of costly subflows (§5.5). Due to the restricted space, we concentrate on an in-depth understanding of a selection of new schedulers that provide significant performance improvements.

5.1 Exploring Redundancy

Multipath TCP originates from the need for higher throughput and reliability. Redundant transmission over different subflows – a crude form of forward error correction – was later proposed and proved helpful for latency-sensitive applications in lossy environments.

The existing redundant scheduler provides full redundancy, i.e., it transmits all packets on all subflows (Fig. 10a, top), unless the packet is already acknowledged and therefore removed from `QU` before being sent on the slower subflow. We argue that the design space for redundant schedulers is widely unexplored. For illustration, we propose two *new* redundant schedulers: `OpportunisiticRedundant` sends packets on all subflows which have not exhausted their congestion window when a packet is *scheduled for the first time* (Fig. 10a, bottom). For thin flows, this scheduler provides full redundancy. Compared to the existing scheduler, however, gradually incoming acknowledgements ensure that the scheduler favors *fresh* packets over the transmission of redundant packets in case the sending queue `Q` fills. We additionally propose `RedundantIfNoQ`, a scheduler that always favors new packets and only retransmits packets in case the sending queue `Q` is empty.

Our programming model enables a rapid evaluation and comparison of these schedulers. Following existing evaluations for the redundant scheduler [17], Fig. 10b shows the average flow completion time (FCT) vs. the flow size (2 subflows, 2% loss in Mininet [23]). In the evaluation, all redundant schedulers outperform the default scheduler for small flows. For increasing flow sizes, the `OpportunisiticRedundant` scheduler beats the existing redundant scheduler as full redundancy becomes more expensive. Our `RedundantIfNoQ` scheduler, that selectively deploys redundancy without delaying fresh packets, outperforms all depicted schedulers.

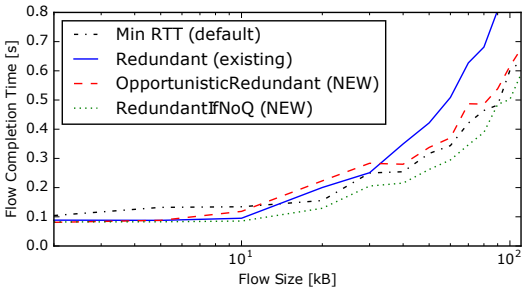
Due to space restrictions, we skip showing the transmission overhead evaluation that basically leads to the same performance

Alexander Frömmgen, Amr Rizk, Tobias Erbshäußer, Max Weller, Boris Koldehofe, Alejandro Buchmann, Ralf Steinmetz
A Programming Model for Application-defined Multipath TCP Scheduling In: Proceedings ACM/IFIP/USENIX Middleware, December 2017

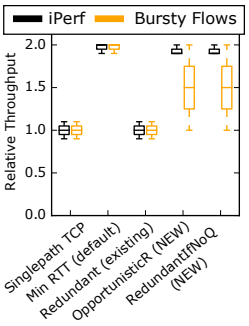

```
Redundant (EXISTING)
1  VAR sbfCandidates = ...
2  FOREACH(VAR sbf IN sbfCandidates) {
3    VAR skb = QU.FILTER(s => !s.SENT_ON(sbf)).TOP;
4    /* are all QU packets sent on this sbf? */
5    IF(skb != NULL) {
6      sbf.PUSH(skb);
7    } ELSE {
8      sbf.PUSH(Q.POP());
9    }
10 }

Redundant (NEW)
2  IF(!sbfCandidates.EMPTY) {
3    FOREACH(VAR sbf IN sbfCandidates) {
4      sbf.PUSH(Q.TOP);
5    }
6    DROP(Q.POP());
7  }
```

(a) Specification of the existing redundant and a new `OpportunisiticRedundant` scheduler.



(b) Average FCT vs. flow size with two homogenous subflows and 2% loss in Mininet.



(c) Throughput normalized to single path TCP.

Figure 10. Exploration of different redundant MPTCP scheduling strategies.

ranking. Instead, we focus on the maximum achievable throughput (Fig. 10c). Observe that for constantly high bandwidth transfers (iPerf), both new schedulers provide nearly the maximum achievable throughput. For bursty flows, however, the throughput depends on fine timing aspects. Both new schedulers base their behavior on implicit information, e.g., queue sizes and incoming acknowledgements. For example, the `RedundantIfNoQ` scheduler may send packets redundantly just before new data arrives in `Q`.

Our programming model enables the convenient specification of two new schedulers. The `OpportunisiticRedundant` scheduler outperforms the existing redundant scheduler. The `RedundantIfNoQ` scheduler outperforms *all* existing MPTCP schedulers in an evaluation with short data flows and lossy subflows.

5.2 Handover-Aware Scheduler

An important question with regard to the dynamics of MPTCP is the handling of newly established subflows. A prominent example is the establishment of a new subflow during the handover, e.g., from WiFi to cellular as a consequence of a WiFi connection break [38] or proactively [18] before the WiFi connection breaks. Assuming that the mobile devices establishes the cellular subflow for a handover from WiFi, a handover-aware scheduler might aggressively retransmit all packets of the WiFi subflow to compensate WiFi losses. In [11], the authors propose and evaluate comparable approaches independently of this work. Due to space limitations, we skip a detailed evaluation of this approach and focus on more examples to illustrate the expressiveness of our programming model.

5.3 Signaling to Boost Short Flows

In this section, we discuss how application-awareness enables informed redundancy to avoid the previously highlighted performance degradations for bursty flows.

The significance of MPTCP scheduling decisions increases with subflow heterogeneity, e.g., in terms of RTT. Specifically, flow completion times of short flows may suffer significantly in heterogeneous environments. Fig. 11 illustrates a typical situation at the end of a short flow, where the sending queue `Q` is empty and packets are still in flight. Here, we do not assume any particular packet scheduling, i.e., packet *Seq 35* may have been scheduled on *subflow 2* due to an exhausted congestion window at *subflow 1*. In this simplified example, packet *Seq 35* will, in most cases, dominate the

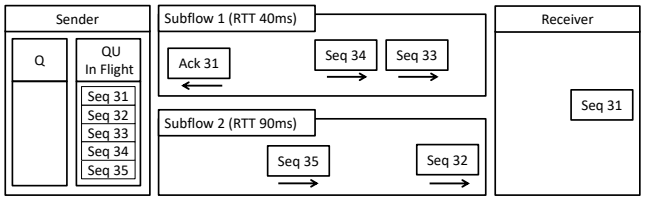


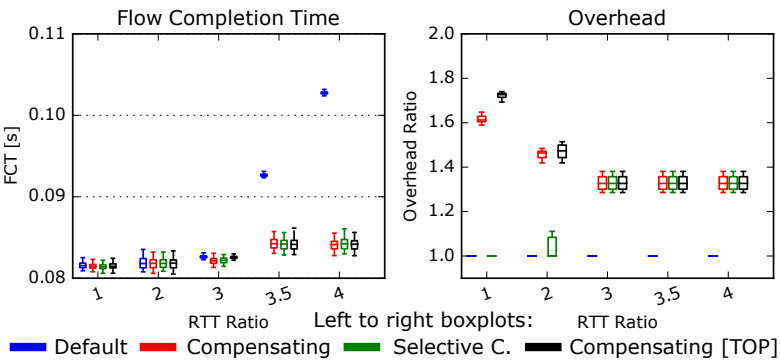
Figure 11. The end of a short flow: The impact of heterogeneous subflow delays on the FCT is most significant when scheduling the last packets of the flow.

FCT. However, entirely avoiding the slower subflow is not optimal either, e.g., the FCT benefits from *Seq 32* on the slower subflow.

The relative ratio of the subflow RTTs, denoted hereafter RTT ratio, is inherently dynamic in communication networks due to queuing, mobility and interface-specific properties. This was empirically shown, for example, for mobile networks [12, 13]. Our programming model enables MPTCP schedulers that precisely leverage application information to address subflow heterogeneity. Next, we show how signaling the *end of flow* by the application leads to improved scheduler performance and optimized FCT for heterogeneous subflows. We propose a `Compensating` scheduler that uses this elementary information to compensate for previous scheduling decisions by retransmitting all packets in flight on subflows where the packets were not sent so far at the signaled end of the flow. Thus, the `Compensating` scheduler retransmits the unacknowledged packets *Seq 32, 35* on *subflow 1* and *Seq 33, 34* on *subflow 2*.

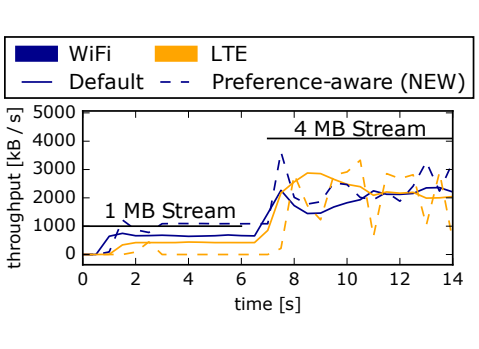
We evaluate the performance of the `Compensating` scheduler, as specified in Fig. 12 (without the highlighted parts), using a Mininet setup with two heterogeneous subflows while varying the RTT ratio. As a baseline, we consider measurements for the default scheduler, where the FCT rapidly increases under high RTT ratios. Being aware of the flow end, the `Compensating` scheduler efficiently retains the FCT under skewed RTT ratios. Our programming model enables a rapid comparison of different flavors of the `Compensating` scheduler. A variation of the choice of the retransmitted packet using `TOP` instead of `FIRST` (on line 15 in Fig. 12) showed only minor impact on the FCT.

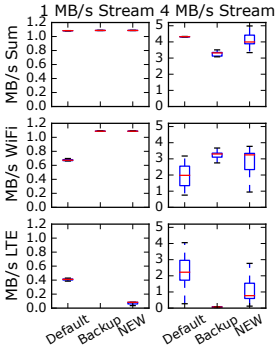
Note that the overhead arising from retransmitting packets in flight (Fig. 12 middle, normalized to the default scheduler) decreases




```
1  DEFINE R1 as endOfFlow;
2  VAR minRttRatio = 2;
3  VAR sbfs = SUBFLOWS.FILTER(sbf =>
4    !sbf.TSQ_THROTTLED AND !sbf.LOSSY AND
5   sbf.CWND > sbf.SKBS_IN_FLIGHT + sbf.QUEUED);
6
7   /* … default scheduler */
8
9  IF(endOfFlow == 1 AND Q.EMPTY) {
10   VAR minRtt = SUBFLOWS.MIN(sbf => sbf.RTT).RTT;
11   VAR maxRtt = SUBFLOWS.MAX(sbf => sbf.RTT).RTT;
12   IF(maxRtt < minRtt * minRttRatio) { RETURN; }
13
14   VAR skbCandidate = QU.FILTER(skb => !sbfs.FILTER(
15           sbf => !skb.SENT_ON(sbf)).EMPTY ).FIRST;
16   sbfs.FILTER(sbf => !skbCandidate.SENT_ON(sbf)).
17           MIN(sbf => sbf.RTT).PUSH(skbCandidate);
18 }
```

Figure 12. Leveraging application information to mitigate the impact of subflow heterogeneity on short flows: The flow-end aware `Compensating` scheduler retains the flow completion time under increasing RTT ratio. Compared to the default scheduler it trades FCT for transmission overhead (normalized to the default scheduler). The highlighted `Selective Compensation` scheduler balances FCT benefits and the transmission overhead.






```
1  DEFINE R1 AS targetBwKB, R4 AS prefAhead;
2  VAR factor = 100, maxAhead = 100 * factor;
3
4  VAR sbfs = SUBFLOWS.FILTER(s => s.SKBS_IN_FLIGHT +
5    s.QUEUED < s.CWND AND !s.LOSSY);
6  VAR prefS = sbfs.FILTER(s=>!s.BACKUP).MIN(s=>s.RTT);
7  IF (prefS != NULL) {
8    IF(!prefS.TSQ_THROTTLED) {
9      prefS.PUSH(Q.POP());
10     IF (prefAhead < maxAhead) {
11       SET(prefAhead, prefAhead + factor); } }
12 } ELSE {
13   VAR capKB = prefS.CWND / prefS.RTT_MS * prefS.MSS;
14   VAR ratio = factor * capKB / (targetBwKB - capKB);
15   IF(prefAhead > ratio AND cap < targetBwKB) {
16     sbfs.FILTER(s => !s.TSQ_THROTTLED).
17       MIN(s => s.RTT).PUSH(Q.POP());
18     SET(prefAhead, prefAhead - ratio); } }
```

Figure 13. Evaluation in the wild: In contrast to the default scheduler, the throughput- and preference-aware (`TAP`) scheduler uses the signaled target throughput (1MB/s then 4MB/s) and efficiently utilizes subflows according to their preferences. The highlighted parts ensure that only the required bandwidth is utilized on the non-preferred LTE subflow. Relying on the existing backup mode shows that WiFi cannot solely sustain the required throughput.

for higher RTT ratios. Equipped with the comparison of the default and the `Compensating` scheduler, we present a `Selective Compensation` scheduler (highlighted parts in Fig. 12) which is tuned for compensating behavior only for RTT ratios > 2. Obviously, this parameter is application and scenario specific. Fig. 12 shows that the `Selective Compensation` scheduler efficiently balances the FCT benefits vs. the overhead at different RTT ratios.

Here, we showed that the extended scheduling API enables new schedulers that significantly improve the FCT in heterogeneous environments with *informed* redundancy. Furthermore, we utilized our programming model to selectively tweak schedulers.

5.4 Combining Application- and Preference-Awareness

We now consider MPTCP schedulers that combine application- and preference-awareness to retain performance targets such as deadlines, required throughput or tolerable RTTs. Subflow preferences might capture a wide range of prioritizations, e.g., to consider asymmetric subflow costs between data-centers or to avoid over-utilizing metered mobile links.

Target Deadline The class of deadline-driven applications such as *Dynamic Adaptive Streaming over HTTP* (DASH) pose deadlines on the arrival times of data chunks. Here, a preference-aware scheduler restricts scheduling packets on non-preferred subflows except if required to retain data chunk deadlines. The very recent MP-DASH [22], which operates, however, on top of the default scheduler, comprises the first scheduler to include such preference-awareness so far. Our programming model straightforwardly captures the deadline driven scheduling functionality while capitalizing on the ability of a fine-grained scheduler specification and timely subflow information.

Target Throughput For applications which require a constant bitrate stream, e.g., interactive video applications, throughput variations are detrimental to the Quality of Experience (QoE). Revisiting our motivating example in Fig.1, our programming model enables specifying a preference-aware scheduler that only resorts to non-preferred subflows if the target throughput is not achieved. Fig. 13 shows the specification of the `TAP` scheduler where the application signals the required minimum throughput to the scheduler by setting the `targetBw` in register `R1` accordingly. The scheduler uses the up-to-date subflow properties *per scheduling decision* to

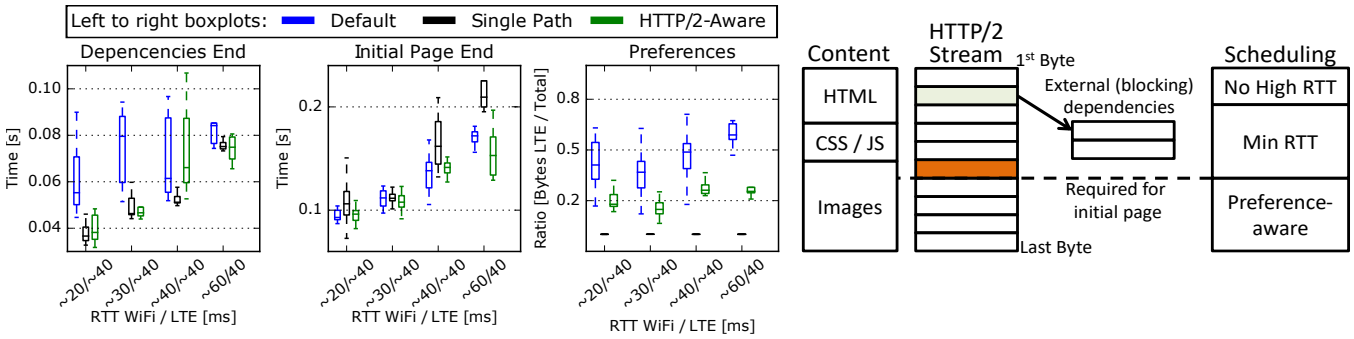


Figure 14. Real-world evaluation of a novel HTTP/2-aware MPTCP scheduler that laverages the HTTP/2 content retrieval process and significantly saves on metered LTE connection. The time to retrieve all dependency information is significantly reduced by avoiding high RTT subflows for the initial packets without affecting the remaining time for non-external content. The preference-aware scheduling of content which is **not required** for the initial page view significantly reduces the usage of the metered LTE subflow.

calculate the expected throughput. Non-preferred subflows (in this case LTE) are only used if required and are restricted to transmit the leftover fraction $\frac{\text{targetBw}-\text{capacityPrefered}}{\text{targetBw}}$ (highlighted parts).

We evaluated the TAP scheduler in the wild between a client Laptop and an Amazon EC2 server instance both running our runtime environment. On the client side we used a Nexus 5 device with USB tethering as LTE modem to a major European service provider and an IEEE 802.11n WiFi connection to a typical residential broadband service. Fig. 13 shows that, compared with the existing default scheduler, TAP reduces the non-preferred LTE usage to a minimum while sustaining the required overall stream throughput. Observe that the TAP scheduler deals very efficiently with fluctuations in WiFi throughput. Note that using our scheduler specification we can tune many flavors of throughput- and application-aware schedulers for various throughput sensitive applications.

Target RTT A scheduler that retains a maximum RTT greatly improves the QoE for interactive applications such as voice-based personal assistant systems where request-response transmissions usually consist of a few packets. Similar to the aforementioned target schedulers, backup subflows can be selectively utilized to retain RTTs below a given target. Due to space restrictions we skip a detailed evaluation of latency- and preference-aware schedulers. However, to illustrate the strong potential of such schedulers we refer to a recent massive, international measurement study [13] of multi-homed wireless MPTCP performance over WiFi and LTE interfaces. The authors of [13] showed that around 15% of all measurement samples experienced a significantly higher RTT on WiFi compared with LTE. A latency- and preference-aware scheduler that is concisely expressed similar to Fig. 13 enables a substantial improvement of the performance of interactive applications in such scenarios while preserving subflow preferences.

5.5 Towards HTTP/2-aware Scheduling

A significant boost to mobile web performance is a key to *universal* MPTCP deployment. Next, we analyze relevant aspects of today's web infrastructure and implement a novel HTTP/2-aware scheduler that overcomes existing limitations.

Analysis Today, browsers, web servers, and the web content are highly tuned, resulting in complex dependencies. With HTTP/2, browsers may, for example, signal priorities to web servers to favor CSS and JavaScript over images to reduce initial loading times [5]. Parsing the resulting content stream may trigger additional requests to 3rd-party content (3PC) as depicted in Fig. 14 (right). One fourth of the Alexa-200 pages have 3PC dependencies on their critical path of initial page loading [52].

A recent measurement study using off-the-shelf MPTCP [21] showed that web protocols incur complex interactions with the multipath-enabled transport layer. Thus, web content optimizations for single path TCP may result in performance degradation for MPTCP. Likewise, the transition from HTTP/1.1 to the emerging HTTP/2 protocol particularly improves performance for pages that rely on a single TCP connection.

We highlight two shortcomings of today's uninformed MPTCP schedulers: *(i)* packets which refer to external resources may be scheduled on slow subflows, delaying the corresponding 3PC requests and thereby potentially increasing the page load time and *(ii)* subflow preferences are not considered. This leads to transferring substantial data amounts, e.g., images, on non-preferred subflows, such as metered LTE, *after* the initial page is loaded, hence, not improving the user QoE.

HTTP/2-aware Scheduling To overcome these shortcomings we implemented a novel HTTP/2-aware scheduler. This scheduler leverages all building blocks of our programming model, using content-dependent scheduling strategies (Fig. 14 right). For the initial data, i.e., until the information on external dependencies is transferred, subflows with high RTTs are avoided by the scheduler. Therefore, the scheduler relies on additional information provided by the webserver. The remaining data that is required to render the initial page is transferred with the default minimum RTT scheduler strategy. For the additional data, that is not required for the initial page and therefore independent of the user QoE, we invoke preference-awareness.

MPTCP-aware Webserver We extended the Nghttp2 library [1] to forward HTTP information through the OpenSSL library to our scheduler API. Each packet is annotated with the content type of the contained HTTP data. The scheduler registers contain information about the number of required bytes for the initial page. As MPTCP supports legacy applications, TLS is supported on top off MPTCP without modification. Note that content type dependent scheduling of encrypted content is an information disclosure.

Evaluation We conducted a measurement study in the wild, using our MPTCP-aware web server on an EC2 instance and a recent

Alexander Frömmgen, Amr Rizk, Tobias Erbshäußer, Max Weller, Boris Koldehofe, Alejandro Buchmann, Ralf Steinmetz
A Programming Model for Application-defined Multipath TCP Scheduling In: Proceedings ACM/IFIP/USENIX Middleware, December 2017

Google Chrome browser on our client from (§5.4) with a WiFi and an LTE interface. To evaluate the impact of the RTT ratio, we systematically increased packet delays for the WiFi interface. Inspired by major optimized web pages, such as `amazon.com`, we similarly used an example page with optimized HTML and JavaScript layout. Thus, more than half of the data, in particular images that lie outside of the initial view, are transferred after the initial page. Our measurements show that the HTTP/2-aware scheduler successfully tunes the initial dependency retrieval time for heterogeneous RTTs (Fig. 14), and therefore enables *earliest possible dependency resolution*. The preference-aware scheduling efficiently reduces the transferred data on the less preferred LTE subflow without affecting the initial page load time.

A systematic improvement of the *overall* HTTP stack is far beyond the scope of this paper. However, by leveraging our programming model, we effectively showed significant improvements overcoming limitations of today's MPTCP schedulers with HTTP/2.

6 Discussion and Limitations

In this section, we discuss target users, design space restrictions, and limitations of our programming model, before providing a pathway to go beyond MPTCP.

Target Developer Our programming model enables network administrators of a managed network environment to specify schedulers that incorporate network specifics and optimize in accordance to additional management functions, such as explicit path controls [26]. For unmanaged networks, e.g., for deployments facing users over the Internet, we anticipate that in particular developers of applications that crucially depend on network performance characteristics will specify an own application-aware scheduler. Moreover, extended application libraries may ensure that entire application classes benefit from improved schedulers. The presented HTTP/2-aware scheduler provides a first step in this direction. Note that although the scheduler runs in the operating system, the interpreter and the JIT-based execution environment enable individual schedulers per application in multi-tenancy and light-weight container environments such as `Docker`.

Usability Kernel knowledge is not required to use the well-known `TC` and `route` configurations. Similarly, our programming model bridges the gap between profound Kernel knowledge, networking details and the application logic. It provides type-safety and handles failures gracefully. In contrast, slight thoughtlessness regarding the disappearance of a subflow in a C Kernel module scheduler easily results in Kernel panics crashing the operating system. As the number of lines required to express schedulers is much lower than equivalent implementations in C, we argue that the presented scheduler model is more usable than any existing approach. We consider a detailed usability study beyond the scope of this paper. Yet, we conducted a limited user study with computer science students where 9 out of 10 participants, who never modified Kernel sources, successfully implemented a new scheduler with predefined functionality using our programming model.

Timeliness vs. Expressiveness Our programming model is designed for efficient and timely MPTCP scheduling decisions taken in the networking stack. More complex and time-consuming computations, e.g., solving optimization problems or forecasting models to control scheduling decisions, should be executed *outside* of the networking stack. Therefore, the *in-Kernel* scheduler is easily configurable by an external controller, e.g., using our extended scheduling API. This is comparable to today's SDN environments, where evaluating forwarding rules is extremely fast whereas time-consuming operations are executed in the SDN controller. The inability of the programming model to express such computations is intended to balance timeliness and expressiveness. A detailed comparison with programming models of other domains is out of this papers scope. We note, however, that eBPF has different and partially even more strict restrictions for programs pushed from the userland. While eBPF does not support loops [9] to ensure termination, our programming model allows `FOREACH` loops.

Dependencies The state-of-the-art MPTCP implementation separates concerns, i.e., congestion control, scheduling, and path management. Our programming model makes the dependencies between these concerns explicit. This enables a deeper analysis of their interactions and provides the abstraction for *cross-concern* optimizations, e.g., jointly considering functionalities of congestion control and scheduling. The scheduler could, for example, relax the congestion window constraint `sbf.CWND > sbf.SKBS_IN_FLIGHT` for the last few N packets of a flow to save an RTT.

Going Beyond MPTCP Currently our programming model is confined to MPTCP, the defacto protocol utilizing multipath functionality as of today. The concepts of our programming model are merely the seed for what we envision as a generalized and flexible programming model for multipathing. For example, the illustrated concepts could be applied to protocols for UDP multipathing, which have different target optimizations and a naturally reduced set of subflow properties. Consequently, we plan to extend the programming model for QUIC [20], e.g., by considering additional semantics, such as, for HTTP frames. A generalized programming model would depart with the in-order delivery property of MPTCP which enables optimizations over out-of-order TCP concepts [34, 36] and pushes the model towards a generalized resource allocation perspective with objectives such as priority or rate-controlled scheduling.

7 Conclusion

This paper presented a high-level programming model for Multipath TCP schedulers that allows a convenient and concise specification of a wide range of application-defined schedulers. We implemented an efficient runtime environment for the Linux Kernel, which provides isolation between schedulers of concurrently executed applications. With the ability of a direct execution of specified schedulers we close the gap between specification and deployment. In our evaluation, we implemented novel, bespoke schedulers which outperformed existing schedulers in network emulations and real-world measurements. In particular, we explored and evaluated scheduler designs that *(i)* leverage transmission redundancy in lossy environments, *(ii)* use application signaling to boost short flows and *(iii)* benefit from application- and preference-awareness for interactive streaming and HTTP/2 content retrieval.

This paper provides the tools necessary for MPTCP scheduler innovations. Our implementation is publicly available at https://progmp.net in the hope to fuel future innovations going beyond the scope of the schedulers presented here.

Acknowledgment

This work has been funded by the German Research Foundation (DFG) as part of the projects B4 and C2 in the Collaborative Research Center (SFB) 1053 MAKI. This work was supported by the AWS Cloud Credits for Research program.

References

[1] Nghttp2: HTTP/2 C Library. https://nghttp2.org/, 2015.

[2] A. Alim, R. G. Clegg, L. Mai, L. Rupprecht, E. Seckler, P. Costa, P. Pietzuch, A. L. Wolf, N. Sultana, J. Crowcroft, et al. Flick: developing and running application-specific network services. In *USENIX ATC*, pages 1–14, 2016.

[3] B. Arzani, A. Gurney, S. Cheng, R. Guerin, and B. T. Loo. Impact of path characteristics and scheduling policies on MPTCP performance. In *IEEE WAINA*, pages 743–748, 2014.

[4] S. Barre, C. Paasch, and O. Bonaventure. Multipath TCP: From theory to practice. In *IFIP Networking*, pages 444–457, May 2011.

[5] M. Belshe, M. Thomson, and R. Peon. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, 2015.

[6] N. Cardwell, Y. Cheng, L. Brakmo, M. Mathis, B. Raghavan, N. Dukkipati, H.-k. J. Chu, A. Terzis, and T. Herbert. packetdrill: Scriptable network stack testing, from sockets to packets. In *USENIX ATC*, 2013.

[7] G. Chen, Y. Lu, Y. Meng, B. Li, K. Tan, D. Pei, P. Cheng, L. L. Luo, Y. Xiong, X. Wang, et al. Fast and cautious: Leveraging multi-path diversity for transport loss recovery in data centers. In *USENIX ATC*, 2016.

[8] Y.-C. Chen, Y.-s. Lim, R. J. Gibbens, E. M. Nahum, R. Khalili, and D. Towsley. A measurement-based study of multipath TCP performance over wireless networks. In *ACM IMC*, pages 455–468, 2013.

[9] J. Corbet. Extending extended BPF. https://lwn.net/Articles/603983/, 2014.

[10] X. Corbillon, R. Aparicio-Pardo, N. Kuhn, G. Texier, and G. Simon. Cross-layer scheduler for video streaming over MPTCP. In *ACM MMSys*, 2016.

[11] Q. De Conick and O. Bonaventure. Every Millisecond Counts: Tuning Multipath TCP for Interactive Applications on Smartphones. http://hdl.handle.net/2078.1/185717, 2017.

[12] Q. De Coninck, M. Baerts, B. Hesmans, and O. Bonaventure. A first analysis of Multipath TCP on smartphones. In *PAM*, pages 57–69. Springer, 2016.

[13] S. Deng, R. Netravali, A. Sivaraman, and H. Balakrishnan. WiFi, LTE, or both?: Measuring multi-homed wireless internet performance. In *ACM IMC*, pages 181–194, 2014.

[14] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing web latency: the virtue of gentle aggression. *SIGCOMM Computer Communication Review*, 43(4):159–170, 2013.

[15] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824.

[16] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ACM ICFP*, pages 279–291, 2011.

[17] A. Froemmgen, T. Erbshaeuser, T. Zimmermann, K. Wehrle, and A. Buchmann. ReMP TCP: Low latency multipath TCP. In *IEEE ICC*, 2016.

[18] A. Froemmgen, S. Sadasivam, S. Mueller, A. Klein, and A. Buchmann. Poster: Use your senses: A smooth multipath tcp wifi/mobile handover. In *ACM MobiCom*, pages 248–250, 2015.

[19] Y. Go, O. C. Kwon, and H. Song. An energy-efficient HTTP adaptive video streaming with networking cost constraint over heterogeneous wireless networks. *IEEE Transactions on Multimedia*, 17(9):1646–1657, 2015.

[20] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk. QUIC: A UDP-based secure and reliable transport for HTTP/2, July 2016. IETF, Internet-Draft.

[21] B. Han, F. Qian, S. Hao, L. Ji, and N. Bedminster. An anatomy of mobile web performance over Multipath TCP. In *ACM CoNEXT*, 2015.

[22] B. Han, F. Qian, L. Ji, and V. Gopalakrishnan. MP-DASH: Adaptive video streaming over preference-aware multipath. In *ACM CoNEXT*, 2016.

[23] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *ACM CoNEXT*, 2012.

[24] B. Hesmans and O. Bonaventure. An enhanced socket API for Multipath TCP. In *IRTF Applied Networking Research Workshop*, 2016.

[25] B. Hesmans, G. Detal, R. Bauduin, O. Bonaventure, et al. SMAPP: Towards smart Multipath TCP-enabled applications. In *ACM CoNEXT*, 2015.

[26] S. Hu, K. Chen, H. Wu, W. Bai, C. Lan, H. Wang, H. Zhao, and C. Guo. Explicit path control in commodity data centers: Design and applications. In *USENIX NSDI*, 2015.

[27] J. Hwang, A. Walid, and J. Yoo. Fast coupled retransmission for multipath TCP in data center networks. *IEEE Systems Journal*, 2016.

[28] R. Khalili, N. Gast, M. Popovic, U. Upadhyay, and J.-Y. Le Boudec. MPTCP is not pareto-optimal: performance issues and a possible solution. In *ACM CoNEXT*, 2012.

[29] E. Kohler, M. F. Kaashoek, and D. R. Montgomery. A readable TCP in the prolac protocol language. In *ACM SIGCOMM*, 1999.

[30] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

[31] N. Kuhn, E. Lochin, A. Mifdaoui, G. Sarwar, O. Mehani, and R. Boreli. DAPS: intelligent delay-aware packet scheduling for multipath transport. In *IEEE ICC*, 2014.

[32] I. Lopez, M. Aguado, C. Pinedo, and E. Jacob. SCADA systems in the railway domain: Enhancing reliability through redundant multipathtcp. In *IEEE ITSC*, 2015.

[33] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[34] S. McQuistin, C. Perkins, and M. Fayed. TCP hollywood: An unordered, time-lined, TCP for networked multimedia applications. In *IFIP Networking*, pages 422–430, 2016.

[35] A. Nikravesh, Y. Guo, F. Qian, Z. M. Mao, and S. Sen. An in-depth understanding of multipath TCP on mobile devices: Measurement and system design. In *ACM MobiCom*, pages 189–201, 2016.

[36] M. F. Nowlan, N. Tiwari, J. Iyengar, S. O. Amin, and B. Ford. Fitting square pegs through round pipes: Unordered delivery wire-compatible with TCP and TLS. In *USENIX NSDI*, pages 28–28, 2012.

[37] C. Paasch and S. Barre. Multipath TCP in the Linux Kernel. available from http://www.multipath-tcp.org.

[38] C. Paasch, G. Detal, F. Duchene, C. Raiciu, and O. Bonaventure. Exploring mobile/wifi handover with Multipath TCP. In *ACM SIGCOMM Workshop on Cellular networks: operations, challenges, and future design*, pages 31–36, 2012.

[39] C. Paasch, S. Ferlin, O. Alay, and O. Bonaventure. Experimental evaluation of Multipath TCP schedulers. In *ACM SIGCOMM Workshop on Capacity Sharing*, pages 27–32, 2014.

[40] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, et al. The design and implementation of Open vSwitch. In *USENIX NSDI*, pages 117–130, 2015.

[41] M. Popovici and C. Raiciu. Exploiting multipath congestion control for fun and profit. In *ACM HotNets*, pages 141–147, 2016.

[42] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with Multipath TCP. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 266–277, 2011.

[43] C. Raiciu, M. Handly, and D. Wischik. Coupled congestion control for multipath transport protocols. RFC 6356, 2011.

[44] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How hard can it be? Designing and implementing a deployable Multipath TCP. In *USENIX NSDI*, 2012.

[45] M. Rajiullah, P. Hurtig, A. Brunstrom, A. Petlund, and M. Welzl. An evaluation of tail loss recovery mechanisms for TCP. *ACM SIGCOMM Computer Communication Review*, 45(1):5–11, 2015.

[46] M. Scharf and S. Kiesel. Head-of-line blocking in TCP and SCTP: Analysis and measurements. In *IEEE GLOBECOM*, 2006.

[47] S. Seo. KT's GiGA LTE. https://www.ietf.org/proceedings/93/slides/slides-93-mptcp-3.pdf, 2015.

[48] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM*, pages 15–28, 2016.

[49] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable packet scheduling at line rate. In *ACM SIGCOMM*, pages 44–57, 2016.

[50] V.-H. Tran, Q. De Coninck, B. Hesmans, R. Sadre, and O. Bonaventure. Observing real Multipath TCP traffic. *Computer Communications*, 2016.

[51] O. Traub, G. Holloway, and M. D. Smith. Quality and speed in linear-scan register allocation. In *ACM PLDI*, 1998.

[52] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with WProf. In *USENIX NSDI*, pages 473–485, 2013.

[53] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *USENIX NSDI*, 2011.

[54] H. Xu and B. Li. RepFlow: Minimizing flow completion times with replicated flows in data centers. In *IEEE INFOCOM*, pages 1581–1589, 2014.