Experimental Evaluation of Multipath TCP Schedulers

Christoph Paasch[1], Simone Ferlin[2], Ozgu Alay[2] and Olivier Bonaventure[1]

[1]ICTEAM, UCLouvain, Belgium

[2]Simula Research Laboratory, Fornebu, Norway

ABSTRACT

Today many end hosts are equipped with multiple interfaces. These interfaces can be utilized simultaneously by multipath protocols to pool resources of the links in an efficient way while also providing resilience to eventual link failures. However how to schedule the data segments over multiple links is a challenging problem, and highly influences the performance of multipath protocols.

In this paper, we focus on different schedulers for Multipath TCP. We first design and implement a generic modular scheduler framework that enables testing of different schedulers for Multipath TCP. We then use this framework to do an in-depth analysis of different schedulers by running emulated and real-world experiments on a testbed. We consider bulk data transfer as well as application limited traffic and identify metrics to quantify the scheduler's performance. Our results shed light on how scheduling decisions can help to improve multipath transfer.

Categories and Subject Descriptors

C2.5 [**Computer-communication Networks**]: Local and Wide-Area Networks—*Internet (e.g., TCP/IP)*

Keywords

Scheduling; Experimenting; Multipath TCP

1. INTRODUCTION

Today's Internet is radically different from what it was 30 years ago, the time when the building blocks of the Internet (e.g. TCP and IP) have been specified. At that time, end hosts had a single interface. However, today end hosts often have multiple interfaces to access the world wide web. For example, smartphones are equipped with two interfaces: a WiFi and a mobile broadband (e.g., 3G/4G). Similarly, server machines are multihomed and data center networks have a large redundant infrastructure with many different paths between any two servers. However, TCP is not able to efficiently utilize this multipath infrastructure as it tightly couples the data stream to the source and destination IP addresses used to establish the connection.

CSWS'14, August 18, 2014, Chicago, IL, USA.

http://dx.doi.org/10.1145/2630088.2631977.

Multipath TCP closes this gap between the *multipath network* and the *single-path transport*. Multipath TCP is a major extension to TCP, allowing the use of multiple paths between two end-hosts for the transmission of a single data stream [6]. This enables the pooling of resources as the paths may go over different interfaces with distinct bottlenecks, effectively increasing the goodput for the end user [14, 3]. Further, Multipath TCP permits vertical handover for mobile nodes, offloading traffic from WiFi to 3G [12]. Multipath TCP is best at providing these benefits with long-lasting flows. These flows may be bulk transfers or rate-limited traffic, like media streaming applications. For the latter, Multipath TCP might bring a benefit if a single network connection does not provide sufficient bandwidth or reliability. Here, not only the throughput but also the end-to-end delays, as well as buffer space requirements, become more relevant [2] as these kind of applications are sensitive to delay-jitter. Examples in this direction are the adoption of Multipath TCP in Apple's iOS7 for the Siri application.

There are many factors influencing the performance of Multipath TCP [16, 13]. One of them is the design of the scheduler. The scheduler is responsible for the distribution of data over multiple paths and wrong scheduling decisions might introduce head-of-line blocking or receive-window limitation, especially when paths are heterogeneous. In such a scenario, the user will observe high delays as well as goodput degradation for its application, resulting in poor user experience. Therefore, the scheduler can have a significant impact on the performance of Multipath TCP.

We introduce a modular scheduler framework that allows to easily change the way data is distributed over the subflows. Further, we evaluate different schedulers for Multipath TCP and provide an in-depth performance analysis considering both bulk data transfers and application-limited flows. We consider goodput and application delay as metrics. Our experiments include evaluations with both an emulated environment using the *Experimental Design*-approach [13] and with real WiFi and 3G networks with the NorNet testbed [11]. We identify the impact of scheduling decisions on the performance of Multipath TCP and illustrate the underlying root cause for the observed behavior. We provide guidelines on the properties of a good scheduler to achieve a good performance under different scenarios. The design of such a scheduler is out of the scope and left for future work.

This paper is structured as follows. Section 2 discusses the background on Multipath TCP and the two main constraints that a scheduler needs to take into account. Section 3 describes related work and the schedulers evaluated in this paper. The measurement setup for our evaluations and the experimental results are presented in Section 4. Finally, we conclude our work and discuss the results in Section 5.

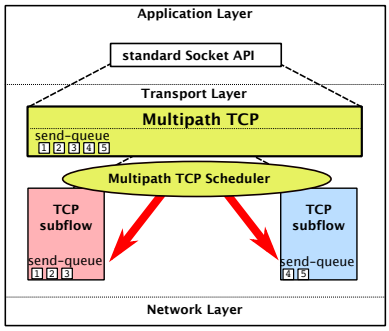


Figure 1: The scheduler distributes the segments from the Multipath TCP level on the different TCP subflows.

2. MULTIPATH TCP

Multipath TCP is a major protocol extension to TCP that supports the transmission of a single data stream across different interfaces (e.g., WiFi and 3G on a smartphone). Multipath TCP increases the goodput for the application by efficiently pooling the network's resources [6]. This pooling is achieved by presenting a regular stream-socket interface to the application, however below this interface, TCP subflows are created for each path. These subflows form together a Multipath TCP connection, using TCP options to signal the necessary control information between the end hosts. These TCP subflows make Multipath TCP look like regular TCP for a firewall/middlebox along the subflows' path. Thus, making Multipath TCP deployable on today's Internet [16].

2.1 Exchanging Data

The pooling of the subflow's resources is achieved by multiplexing individual segments across the different subflows. TCP options are used to allow the receiver to reorder the segments and recreate the byte stream ensuring reliable and in-order delivery. Each subflow is subject to the regular congestion control stages like slow-start and congestion-avoidance. Specific congestion control schemes are used to allow a better load balancing and fairness among the subflows [23, 10].

When multiplexing individual segments, Multipath TCP has to decide on which subflow to schedule each segment. We call the module which takes this decision the *scheduler* in the remainder of this paper. Figure 1 illustrates the architecture of a Multipath TCP implementation, its subflows and the role of the scheduler. The scheduler has access to the state of each TCP subflow, including congestion window and RTT estimation. In the next two sections we describe two of the main constraints a Multipath TCP scheduler needs to consider.

2.2 Head-Of-Line Blocking

The TCP subflows of the Multipath TCP connection may go through paths with different characteristics. For example, a subflow going over the smartphone's WiFi interface experiences a much lower RTT than a subflow sent over the phone's 3G interface.

As packets are multiplexed across the different subflows, assuming each subflow goes on a different path, the path's delay difference might cause out-of-order delivery at the receiver. As Multipath TCP ensures in-order delivery, the packets that are scheduled on the low-delay subflow have to "wait" for the high-delay subflow's packets to arrive in the out-of-order queue of the receiver. This phenomenon is known as *head-of-line blocking* [19].

Head-of-line blocking causes *burstiness* in the data stream by delaying the data delivery to the application, which is undesirable especially for interactive or streaming traffic. Interactive applications will become less reactive, resulting in poor user experience. Streaming applications will need to add a high amount of application-level buffering, stressing the end systems to cope with *burstiness* and provide a continuous streaming experience to the end user.

2.3 Receive-Window Limitations

A TCP stack reserves a certain amount of memory for out-of-order data that might be received in the event of in-network reordering or packet loss. Multipath TCP introduces reordering across the TCP subflows due to the delay differences, hence the receive buffer has to accommodate out-of-order data also at the Multipath TCP level. Thus, the size of the receive buffer is critical to allow high goodput.

In order to fully utilize the capacity of all paths, a receiver must provide enough buffer space so that the sender can keep all subflows fully utilized, even in the event of reordering due to delay differences or loss. The recommendation for Multipath TCP's receive buffer size is defined in [1]:

$$\text{Buffer} = \sum_{i}^{n} \text{bw}_i \times \text{RTT}_{\max} \times 2$$

where each subflow will be able to send at full speed ($\sum_i^n \text{bw}_i$) during the time-interval of the highest round-trip-time among all subflows ($\text{RTT}_{\max}$), even if a loss event occurs (multiply by 2).

However, Some end hosts are not able to provide the necessary amount of memory to allocate enough buffer to utilize the full capacity [2]. This phenomenon has been studied in [16, 13], proposing incremental changes to the heuristics within Multipath TCP by retransmitting segments and penalizing slow subflows, detailed in the following section.

3. SCHEDULERS

A wrong scheduling decision might result in head-of-line blocking or receive-window limitation, affecting the performance of Multipath TCP, as discussed in the previous section. Accurately scheduling data across multiple paths while trying to avoid head-of-line blocking or receive-window limitation has been shown to be challenging, in particular if the multiple paths are heterogeneous. How to schedule different SCTP streams across the different SCTP associations has been analyzed in [20]. However, SCTP design is different from Multipath TCP. Standard SCTP does not support the transmission of a single stream across different paths. With [7], concurrent multipath transfer has been introduced for SCTP (SCTP-CMT) and thus exposes SCTP now also to the scheduling problem in a similar manner as Multipath TCP. [18] tries to achieve ordered delivery at the receiver in SCTP-CMT by taking the delay of each path into account. While in theory this is promising, it is unclear how feasible it is in a real-world Linux kernel implementation where only a rough estimate of the path's delay is available.

In this paper, we implemented a modular Multipath TCP scheduler framework[1]. Whenever the stack is ready to send data (e.g., an acknowledgement has freed up space in the congestion window, or the application pushed data into the send-queue), the scheduler is invoked to execute two tasks: first, choose a subflow among the set of TCP subflows and; second, decide which segment to send considering the properties of the subflow. We added callbacks from the Multipath TCP stack that invoke the functions specific to each

[1]The code is available since release v0.89 at http://multipath-tcp.org

```
MPTCP: An Multipath TCP connection, ready to send data.
    MPTCP->sched represents a structure containing the
    specific callbacks.
 1: subflow = MPTCP->sched->get_subflow();
 2: while subflow != NULL do
 3:     data = MPTCP->sched->get_data(subflow);
 4:     while data != NULL do
 5:         send_data(subflow, data);
 6:         data = MPTCP->sched->get_data(subflow);
 7:     end while
 8:     subflow = MPTCP->sched->get_subflow();
 9: end while
```

Figure 2: Pseudocode of the modular scheduler framework, using callbacks to invoke the scheduling functions.

scheduler. A pseudo-code implementation of this behavior can be seen in Figure 2. This allowed us to design the scheduler in a modular infrastructure, as it is done with the TCP congestion control algorithms [22]. A *sysctl* allows to choose the default scheduler for all Multipath TCP connections.

Within our modular framework, we consider different schedulers. First, we discuss a simple round-robin scheduler. Then, considering the heterogeneous networks, where significant delay differences are observed between the subflows, we discuss delay-based schedulers. A first evaluation has been done in [21]. But only a limited environment has been used in the evaluation and improvements to the schedulers [16] were not yet part of the Multipath TCP implementation.

3.1 Round-Robin (RR)

The round-robin scheduler selects one subflow after the other in round-robin fashion. Such an approach might guarantee that the capacity of each path is fully utilized as the distribution across all subflows is equal. However, in case of bulk data transmission, the scheduling is not really round-robin, since the application is able to fill the congestion window of all subflows and then packets are scheduled as soon as space is again available in each subflow's congestion window. This effect is commonly known as *ack-clock* [8].

Such a scheduler has already been discussed for concurrent multipath transfer SCTP [7]. [5] evaluates how such a round-robin scheduler behaves in CMT-SCTP for multi-streaming compared to a scheduler that assigns each stream to a specific path.

3.2 Lowest-RTT-First (LowRTT)

In heterogeneous networks, scheduling data to the subflow based on the lowest round-trip-time (RTT) is beneficial, since it improves the user-experience. It reduces the application delay, which is critical for interactive applications. In other words, the RTT-based scheduler first sends data on the subflow with the lowest RTT estimation, until it has filled its congestion window (as it has been first described in [16]). Then, data is sent on the subflow with the next higher RTT.

In the same way as the round-robin scheduler, as soon as all congestion windows are filled, the scheduling becomes *ack-clocked*. The acknowledgements on the individual subflows open space in the congestion window, and thus allow the scheduler to transmit data on this subflow.

.

As explained in Section 2, the delay difference triggers head-of-line blocking and/or receive-window limitation. Next, we discuss two extensions to the RTT based scheduler. The first solution reacts upon receive-window limitation, and the second solution minimizes the delay difference in the presence of *bufferbloat* on the individual subflows.

3.3 Retransmission and Penalization (RP)

In order to compensate for delay differences, *opportunistic retransmission and penalization* for Multipath TCP have been proposed in [16]. Opportunistic retransmission re-injects the segment causing the head-of-line blocking on the subflow that has space available in its congestion window (similar to chunk rescheduling for CMT-SCTP [4]). This allows to quickly overcome head-of-line blocking situations and compensate for the RTT differences. Further, the penalization algorithm reduces the congestion window of the subflow with the high RTT, hence, reducing the sending rate and the effect of *bufferbloat* on the subflow.

Particularly, the goal is not only to improve goodput, but also to reduce delay, jitter and buffer size requirements.

3.4 Bufferbloat Mitigation (BM)

Following similar observations from Section 3.3, another source for high RTTs are large buffers on routers and switches along the subflow's path. TCP will try to fill these buffers creating *bufferbloat*, resulting in very high RTTs.

The *bufferbloat-mitigation* algorithm is an alternative to the RP mechanism. It caps the RTTs by limiting the amount of data to be sent on each subflow, hence, reducing the effect of *bufferbloat* [17]. The goal here is not to significantly improve goodput, but instead, to improve the application delay-jitter and reduce end host buffer size requirements.

The main idea behind the bufferbloat-mitigation algorithm is to capture *bufferbloat* by monitoring the difference between the minimum smoothed RTT ($sRTT_{min}$), and smoothed RTT (sRTT). Whenever sRTT drifts apart from $sRTT_{min}$ on the same subflow, as a result of sending data, we take it as an indication of *bufferbloat*. Therefore, we cap the congestion window for each subflow by setting an upper bound $cwnd_{limit}$.

$$cwnd_{limit} = \lambda \times (sRTT_{min}/sRTT) \times cwnd.$$

where λ determines the *tolerance* between $sRTT_{min}$ and sRTT. Within the remainder of this paper, we fix λ to 3, which has proven to bring the best results, as analyzed in [17].

4. EVALUATION

Traffic characteristics and the network environment, where the TCP subflows go through, influence the schedulers' performance. This section evaluates two Multipath TCP metrics, namely goodput and application delay-jitter in emulated and real-world experiments.

4.1 Experiment and Emulation Setup

Within `Mininet` the *Experimental Design* approach is used. This experimental design approach, explained in [13], is a structured approach to the evaluation of transport layer protocols. Protocols are influenced by a large number of factors, each of them with a different range of possible values. In case of networking experiments, these influencing factors include (but are not limited to) the propagation delay between client and the server, the capacity of the bottleneck link and the size of the buffer at the bottleneck. When applying the experimental design approach, a domain must be selected for each of these factors. In this paper we use the same domains as in [13] to create low- and a high-BDP environments, shown in Table 1. As a next step, the selection of the parameter

	Low-BDP		High-BDP	
Factor	Min.	Max.	Min.	Max.
Capacity [Mbps]	0.1	100	0.1	100
Propagation delay [ms]	0	50	0	400
Queuing [ms]	0	100	0	2000

Table 1: Domains of the influencing factors for our evaluation of the schedulers within `Mininet`

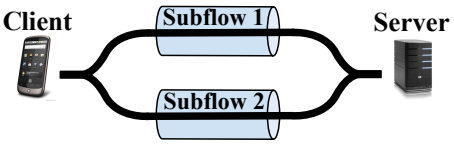


Figure 3: Our testbed sends traffic between two hosts, using two subflows.

sets for the experiments must be done. [13] suggests to use space-filling designs, which equally distribute the points across the whole considered space. We apply the same design for the experiments in this paper, executing up to 200 experiments each for the low- and the high-BDP environments.

The Multipath TCP implementation used in all our evaluations is based on release 0.88[2]. The considered topology consists simply of two hosts communicating to each other over two different subflows (shown in Figure 3), which may experience different path characteristics.

The real-world experiments use the `NorNet` testbed [11]. The `NorNet Edge` testbed runs a standard Linux distribution (3.11.7) and it is meant to do experimental network research on real-world networks. With this testbed, we show exactly the same measurements as defined for the emulated environment with `Mininet`, using real systems and collecting data from operational networks. This is crucial to understand the effect of real network behavior on the Multipath TCP implementation. We consider the smartphone use case where we utilize both WLAN and 3G (UMTS) interfaces to evaluate the performance with heterogeneous links. The WiFi connection uses a public WLAN, connecting ca. 100 people during work hours in a large office complex with several other interfering WLAN networks. On the system level, we continuously flush all cached TCP metrics to avoid any dependency between experiments. Finally, the Olia congestion control [10] is used. Similar results were obtained with the coupled congestion control scheme [23]. Compared to the `Mininet` environments shown in Table 1, the `NorNet` testbed does not offer the same flexibility setting up the measurement environment. Although, for the WLAN network we observe average RTTs of 15 ms and link capacity close to 10 Mbps. However, the 3G (UMTS) network shows average delays from 20 ms up to several seconds (when *bufferbloat* occurs) and capacity of approximately 5 Mbps in downlink.

4.2 Bulk-Transfer

One of the goals of Multipath TCP is to increase the application goodput [15], which can be measured by transferring bulky data between two Multipath TCP capable end hosts.

Mininet.

Within `Mininet` we generate a bulk-transfer using iperf, where each transfer lasts for 60 seconds. Our measurements cover 400 different settings, classified as low-BDP and high-BDP environments (200 settings for each environment). As each of these settings represent a completely different environment, the goodputs of the experiments will be largely different. We need thus to normalize the output to allow comparison of the results among each other. Thus, we measure the *aggregation benefit*, which is a normalization of the benefit in terms of goodput when using Multipath TCP. For example, a value of -1 indicates the minimum (0 Mbps), while 0 means that Multipath TCP achieves the same goodput as regular TCP on the best path, and 1 means that Multipath TCP perfectly aggregates the available capacity. More information about the aggregation benefit metric can be found in [9, 13].

Figure 4 shows the boxplots of the `Mininet` results. The boxplots display the 25th and 75th percentiles, the median as well as the outliers among all 200 experiments. We skip the RR scheduler since it performs similar to LowRTT. This is because in bulk-transfers the TCP subflows are saturated and are thus controlled by the *ack-clock*. Therefore, available space in the congestion window controls the way packets are multiplexed across the subflows rather than the scheduler.

In the low-BDP environment there is no significant difference between schedulers. Each of them achieves close to perfect bandwidth aggregation. Only the RP algorithm improves the worst-case result among the 200 experiments to achieve an aggregation benefit equal to the best available path. In the high-BDP environment receive-window limitations may occur. In this case the RP and BM techniques described in Section 3.3 and 3.4 improve the aggregation benefit. The RP technique has a higher benefit in the lower 25th percentile and the median. This is thanks to the retransmission of the blocking segment(s) as the receive-window limitation in these cases is not due to *bufferbloat* but rather due to the baseline RTT difference.

NorNet.

Within `NorNet Edge` we tested bulk-transfers of 16 MB files in downlink with both bounded (2 MB) and unbounded (16 MB) buffers. The bounded buffers will make the connection more likely to be limited by the receive window. We repeat each measurement around 30 times for each configuration. All measurements are performed in the same networks and at the same locations over a period of 3 weeks.

Figure 5 shows the goodput, for all schedulers. The boxplots display the data among the 30 experiments. For a bulk-transfer with unbounded buffer sizes (16 MB), the goodput of all schedulers is similar. Each scheduler is able to efficiently aggregate the bandwidth of WLAN and 3G together within the `NorNet` testbed. The unbounded buffers allow for sufficiently large memory, so that no receive-window limitation occurs.

However, if the receive buffer is bounded (as it is often the case on a smartphone), the Multipath TCP connection may become receive-window limited. Figure 6 shows the goodput in this case. Here, one can see that LowRTT+BM slightly outperforms the other schedulers. In case of the LowRTT+RP scheduler, the effect of bufferbloat is not optimally reduced by the penalization algorithm, as it does not manage to bring the congestion window sufficiently down so that the delay-difference is reduced. This happens, because the RP algorithm is reactive and does the penalization only after limitation has already happened. The BM algorithm is proactive and, instead, prevents high delay-differences beforehand, achieving higher goodput.

4.3 Application-Limited Flows

This section evaluates the impact of the schedulers on delay-jitter with rate-limited traffic, i.e., when the application is not saturating the connection. In this case, the scheduler has available space in all

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

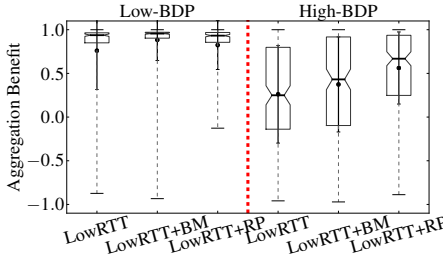


Figure 4: `Mininet`: In high-BDP the connection becomes receive-window limited and BM and RP show their benefits.

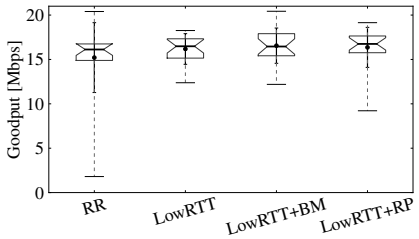


Figure 5: `NorNet`: With unbounded buffers (16MB), each scheduler achieves the goodput.

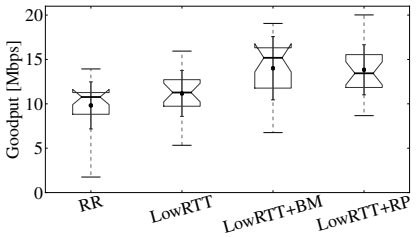


Figure 6: `NorNet`: With bounded buffers (2MB), LowRTT+BM and LowRTT+RP achieve the best performance.

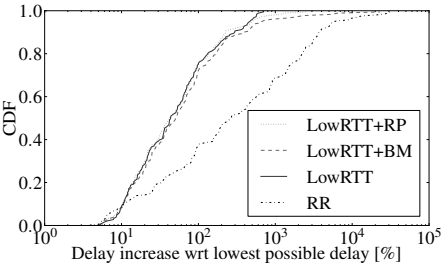
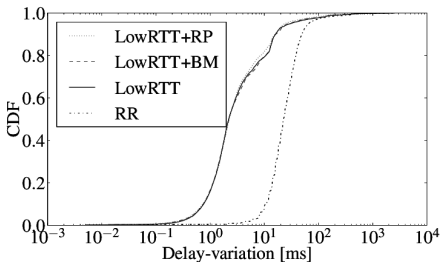
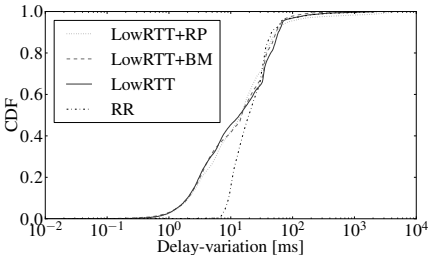


Figure 7: `Mininet`: Using the lowest-RTT-first scheduler greatly reduces the application-delay variance in `Mininet`.



(a) 500 Kbps



(b) 1875 Kbps

Figure 8: `NorNet`: The same delay-increase is observed in our real testbed when using the RR scheduler.

subflows and should select the one that guarantees the lowest delay for each segment.

In order to evaluate the delay-jitter we created an application that sends traffic at a specific rate in blocks of 8 KB. The receiver tracks the timestamps at which each block has been received. As the sender transmits at a constant rate, the receiver is able to detect the delay with respect to the desired packet-arrival time.

Mininet.

Within Mininet we ran this application in the high-BDP environment of the *Experimental Design* approach, for a total of 200 experiments. Figure 7 shows the CDF of the worst-case relative delay increase. The delay increase is expressed in % compared to the lowest one-way delay. E.g., if the minimum one-way delay is 20 ms, but a block of 8 KB has been received rather after 40 ms, the relative delay increase has a value of 100%. We show the worst delay-increase among all 8 KB blocks of each experiment as this will affect the user-experience.

In Figure 7 it is visible that the RR scheduler is particularly bad in terms of application delay. 70% of the experiments using the LowRTT scheduler have a range between 10 and 100% of delay-increase. Using a RR scheduler, roughly 40% of the experiments have a delay-increase between 100 and 1500%. Such delay increases have significant impact on delay-sensitive applications since they would need to maintain large buffers to react upon these delay-spikes.

NorNet.

Within `NorNet` we evaluate the delay-jitter using rate-limited applications transmitting at 500 Kbps and 1875 Kbps in downlink with unbounded buffers. The values for the application-limited rates are at approximately 5 and 10% of the mean goodput of the bulk-transfer. We repeat each measurement around 30 times for each configuration.

Figure 8 shows the variation of the application delay for all schedulers. One can see that in all application-limited scenarios, RR performs mostly worse compared to all other schedulers. More prominent, in Figure 8(a), RR's delay-variance shows to be up to 10 times worse compared to the LowRTT scheduler. This can be explained as RR schedules data without taking the RTT into account. Looking into our dataset, one can see that both subflows carry similar amounts of data, thus, increasing head-of-line blocking when paths have different characteristics, e.g., baseline RTT difference.

For all other schedulers we observe that for very low application rates, see Figure 8(a), LowRTT utilizes mainly one subflow, the subflow with lowest RTTs. This is because the congestion window space is not a limiting factor, and it has mostly enough space to carry all data available.

By increasing the application rate, see Figure 8(b), LowRTT performs in at least 60% of the cases up to 10 times better compared to RR. The remaining 40% can be explained as we see that the subflow with higher RTT (3G network) contributes more compared to the scenario in Figure 8(a). This happens, because at a higher sending rate, the congestion on the WiFi network will make the LowRTT scheduler send traffic on the 3G subflow. This will introduce head-of-line blocking due to the higher delay over the 3G network and thus increases the delay-variation.

We also evaluate the delay-jitter when sending at unlimited rate within the `NorNet` testbed. In this case, both WiFi and 3G are fully utilized and *bufferbloat* might start increasing on the 3G path. We observe that the LowRTT+BM scheduler effectively reduces the bufferbloat and keeps the application delay lower compared to other schedulers.

5. DISCUSSION

In this paper, we have proposed and implemented a modular scheduler selection framework that allows Multipath TCP to change the way data is multiplexed across different TCP subflows. We used this framework to experimentally evaluate schedulers in a wide variety of environments in both emulated and real-world experiments. In these environments, we could quantify the performance of different schedulers, and scheduler extensions, with respect to goodput as well as application delay-jitter.

We discovered that a bad scheduling decision triggers two effects: First, head-of-line blocking if the scheduler sends data across a high-RTT subflow. Second, receive-window limitation, which prevents the subflows from being fully utilized. We have shown that a simple strategy to preferentially schedule data on the subflow with lowest RTT (LowRTT) helps to reduce the application delay-jitter compared to a simple round-robin (RR) scheduler.

Furthermore, the RP extension helps to mitigate receive-window limitation, albeit it is a reactive method, i.e., it tries to recover from receive-window limitation. We have discovered that in some cases this is not sufficient. If the delay-difference is very high due to huge *bufferbloat*, the penalization will not manage to bring the congestion window sufficiently down. Also, anecdotal evidence has shown that the penalization may hurt, if two subflows are unfortunately sent through the same bottleneck. The bufferbloat mitigation technique helps in these cases, but cannot overcome a large difference in the baseline RTT. Further, the delay-based congestion-window capping may also suffer from the known limitations of delay-based congestion controls when the bottleneck is shared with other flows that do not deploy the same window capping.

Multipath scheduling should ideally be done in a way that the data is received in-order. This minimizes head-of-line blocking and receive-window limitation as applications are able to continuously read data out of the receive queue. However, it is not trivial to design such a scheduler with a rough estimation on capacity or RTTs of the paths, maintained by the Linux kernel. In our future work we plan to extend our evaluation framework to a larger set of traffic classes (including cross-traffic and on/off flows). Our modular scheduler framework is publicly available to enable researchers to explore and contribute with other new findings.

6. ACKNOWLEDGMENTS

This research has received funding from the European Union's Seventh Framework Program FP7/2007-2013 under the Trilogy2 project (grant agreement 317756) and the EU project RITE (grant agreement ICT-317700).

7. REFERENCES

[1] S. Barre, C. Paasch, and O. Bonaventure. Multipath TCP: From Theory to Practice. In *IFIP Networking*, 2011.

[2] X. Chen, R. Jin, K. Suh, B. Wang, and W. Wei. Network Performance of Smart Mobile Handhelds in a University Campus WiFi Network. In *ACM IMC*, 2012.

[3] Y.-C. Chen, Y. Lim, R. Gibbens, E. Nahum, R. Khalili, and D. Towsley. A Measurement-based Study of Multipath TCP Performance over Wireless Networks. In *ACM IMC*, 2013.

[4] T. Dreibholz, M. Becke, E.P. Rathgeb, and M. Tuxen. On the Use of Concurrent Multipath Transfer over Asymmetric Paths. In *IEEE GLOBECOM*, 2010.

[5] T. Dreibholz, R. Seggelmann, M. Tüxen, and E.P. Rathgeb. Transmission Scheduling Optimizations for Concurrent Multipath Transfer. In *PFLDNeT*, 2010.

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

[7] J. Iyengar, P. Amer, and R. Stewart. Concurrent Multipath Transfer using SCTP Multihoming over Independent End-to-End Paths. *IEEE/ACM Transactions on Networking*, 2006.

[8] V. Jacobson. Congestion Avoidance and Control. In *ACM SIGCOMM Computer Communication Review*, 1988.

[9] D. Kaspar. *Multipath Aggregation of Heterogeneous Access Networks*. PhD thesis, University of Oslo, 2011.

[10] 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.

[11] A. Kvalbein, D. Baltrūnas, K. Evensen, J. Xiang, A. Elmokashfi, and S. Ferlin-Oliveira. The Nornet Edge Platform for Mobile Broadband Measurements. *Elsevier Computer Networks*, 2014.

[12] C. Paasch, G. Detal, F. Duchene, C. Raiciu, and O. Bonaventure. Exploring Mobile/WiFi Handover with Multipath TCP. In *ACM SIGCOMM workshop CellNet*, 2012.

[13] C. Paasch, R. Khalili, and O. Bonaventure. On the Benefits of Applying Experimental Design to Improve Multipath TCP. In *ACM CoNEXT*, 2013.

[14] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *ACM SIGCOMM*, 2001.

[15] C. Raiciu, M. Handley, and D. Wischik. Coupled Congestion Control for Multipath Transport Protocols. RFC6356, October 2011.

[16] 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.

[17] Ferlin-Oliveira S., Dreibholz T., and ÃŰ Alay. Tackling the Challenge of Bufferbloat in Multi-Path Transport over Heterogeneous Wireless Networks. In *IEEE/ACM IWQoS*, 2014.

[18] G. Sarwar, R. Boreli, E. Lochin, A. Mifdaoui, and G. Smith. Mitigating Receiver's Buffer Blocking by Delay Aware Packet Scheduling in Multipath Data Transfer. In *IEEE WAINA*, 2013.

[19] M. Scharf and S. Kiesel. Head-of-line Blocking in TCP and SCTP: Analysis and Measurements. In *IEEE GLOBECOM*, 2006.

[20] R. Seggelmann, M. Tuxen, and E.P. Rathgeb. Stream Scheduling Considerations for SCTP. In *SoftCOM*, 2010.

[21] A. Singh, C. Goerg, A. Timm-Giel, M. Scharf, and T.-R. Banniza. Performance Comparison of Scheduling Algorithms for Multipath Transfer. In *IEEE GLOBECOM*, 2012.

[22] L. Stewart and J. Healy. Light-Weight Modular TCP Congestion Control for FreeBSD 7. Technical report, CAIA, Tech. Rep, 2007.

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