# SRPT-ECF: challenging Round-Robin for stream-aware multipath scheduling

Baptiste Jonglez* Martin Heusse* Bruno Gaujal*
*Univ. Grenoble Alpes, CNRS, Grenoble INP†, LIG, 38000 Grenoble, France
†Institute of Engineering Univ. Grenoble Alpes

***Abstract*—Multipath TCP has long been the standard multipath transport protocol. However, the recent introduction of Multipath QUIC has changed the landscape by allowing multiple streams to coexist, bringing opportunities for further optimization but also a new set of challenges. New *stream-aware* scheduling algorithms are necessary to account for this new variable.**

**We show that, perhaps counter-intuitively, serving streams using a Round-Robin strategy yields poor performance when looking at stream completion time. We then describe SRPT-ECF, our novel stream-aware multipath scheduling algorithm. We show that our algorithm is optimal in a simple network model and that it exhibits good properties on HTTP/2 traces. We then sketch how it could be implemented within Multipath QUIC to schedule web resources with HTTP/2, paving the way for low-latency multipath HTTP/3 implementations.**

***Index Terms*—multipath scheduler, multipath transport protocol, MPQUIC, HTTP/3**

## I. Introduction

In communication networks, using multiple paths to transmit data is usually beneficial because it can reduce latency, increase throughput and improve reliability. However, achieving good performance using paths with heterogeneous latency is difficult [7]. In particular, it requires a multipath scheduler that judiciously selects which data to transmit on which path.

Multipath TCP [7] (MPTCP) is a standard multipath transport protocol that extends TCP and comes with a comprehensive implementation for Linux. Fig. 1 shows the basic principles of MPTCP: the data stream from the application is packetized and sent out on several paths, and packets are reordered on the receiver side thanks to their sequence numbers.

However, many applications now multiplex several *messages* or *resources* in the same connection: a common example is HTTP/2, where all resources of a web page are multiplexed on the same connection. When such a multiplexed data stream is transported over MPTCP, several issues arise: 1) the scheduler has no visibility on the different sub-streams and may take sub-optimal decisions; 2) the receiver ensures strict reordering and blocks any data that follows a gap: this is called *Head-of-Line blocking* (HoL blocking). But if the blocked data and the gap belong to two unrelated sub-streams, this strict reordering is unnecessary and undesirable because it increases application-to-application latency.

We illustrate these two issues in Fig. 1: 1) a small CSS file is partially scheduled on Path 2 that has a high latency, leading to high overall loading time for this file; 2) the receiver needlessly blocks received data for this CSS file because it is waiting for some data that actually belongs to the other sub-stream. To overcome these issues, it is desirable to use a *stream-aware multipath scheduler* that can use information about each sub-stream to take better decisions. Fig. 2 shows the basic principle of such a scheduler.

A stream-aware multipath scheduler can be implemented on top of a transport protocol such as SCTP or QUIC [5]. Among other design goals, they are both built around a *multi-stream* model, precisely to avoid Head-of-Line blocking between unrelated streams. Stream-aware multipath scheduling was first studied in the context of CMT-SCTP [2], with the early realisation that Round-Robin performed poorly in this case; their proposed solution is to schedule each stream on a single path. In contrast, our proposal allows each stream to possibly use several paths while retaining good performance.

QUIC is originally intended for a single path scenario but provides features such as cryptographic identifiers that make it easy to extend it to the multi-path case, thus enabling Multipath QUIC (MPQUIC) implementations. Practical MPQUIC designs have been proposed [1], [11] but they simply reuse existing MPTCP schedulers and do not explore MPQUIC scheduling in depth. Rabitsch et al. [8] first introduced the concept of *stream-aware scheduling* for MPQUIC; in addition, they propose the SA-ECF scheduler that combines Round-Robin and ECF; Wang et al. [10] also use Round-Robin. A more recent work [12] is closer to our proposal and use similar techniques: the authors take completion time as a metric and avoid Round-Robin. However, our proposal is simpler: in particular, we don't rely on any dependency tree as found in HTTP/2. In addition, we prove that our algorithm has interesting properties. Finally, although it was developed with MPTCP in mind, the DEMS scheduler [4] is similar to our work: indeed, the authors optimize the completion time of atomic *chunks* by ensuring their simultaneous completion on all paths. However, their notion of *chunk* may encompass several application-level messages, because TCP is built around a single stream model; in contrast, MPQUIC provides much more visibility about the application-level message structure to the scheduler. In addition, they simply schedule *chunks* in the order given by the application, and don't consider the more general optimization problem of choosing the order in which *chunks* or messages should be scheduled.

In this article, we explore the scheduling problems arising

This work has been partially supported by the French Ministry of Research project PERSYVAL-Lab under contract ANR-11-LABX-0025-01.

Annex to ISBN 978-3-903176-28-7 ©2020 IFIP

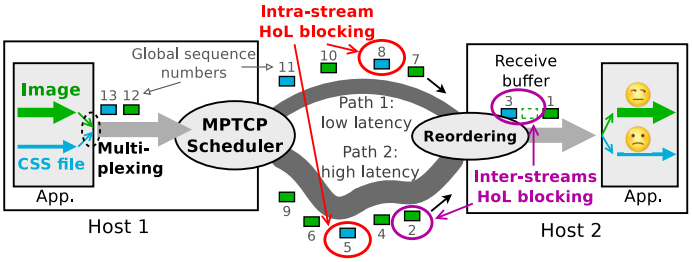


Fig. 1. Multipath TCP basic principle, here with data that has been multiplexed by the application. Head-of-Line blocking can happen both within a stream (packet 8 will arrive before packet 5 for the CSS file) and between streams (packet 3 from the CSS file is waiting for packet 2 from the image).

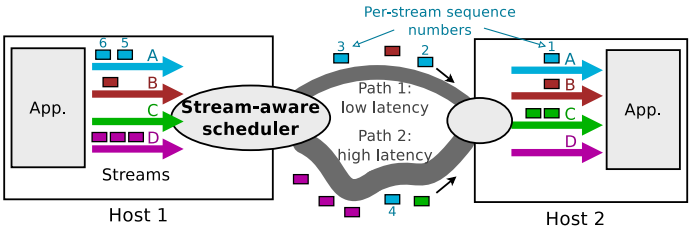


Fig. 2. Stream-aware scheduling, where each stream has its own sequence numbers, thus avoiding inter-stream Head-of-Line blocking. Intra-stream Head-of-Line blocking can be avoided with appropriate scheduling.

from the use of *stream-aware multipath schedulers*: how should streams be ordered? On which paths? What is the best metric to evaluate scheduling strategies? Is there an optimal strategy? Our approach is to use theoretical tools and network models to derive simple yet optimal scheduling strategies, with the hope that it will lead to simpler and better scheduler implementations for real networks.

## II. Single-stream multipath scheduling

We start with the problem of scheduling a single stream on multiple paths, which is exacty the use-case of Multipath TCP. Several schedulers have been proposed [3] besides the original scheduler: we focus on the *Earliest Completion First* [6] (ECF) strategy and show that it is minimizes the completion time for a single message in an idealized network model. We then use this result in Section III to prove the optimality of our proposed algorithm SRPT-ECF in the same model.

The metric we consider is *completion time*, which is the difference between the time at which the stream becomes available and the time at which it has been fully transferred and acknowledged. Other metrics such as *out-of-order latency* [9] are possible, but the completion time is most relevant because we consider streams as atomic messages — that is, we assume that only the completion of the whole stream matters to the application and that the completion of individual bytes within the stream is not directly relevant. This matches the use-case of web resources transferred through HTTP/2 and can even accommodate more complex use-cases such as progressive images: they can be modeled with several atomic messages.

Our network model is the following: each path $p$ has a fixed round-trip latency $D_p$ and a fixed bottleneck capacity or *rate* $R_p$, expressed in bytes per second. This model is very simple but captures the two main parameters of interest for a transport protocol. Even though using fixed values for these parameters is widely unrealistic, it will allow us to more clearly highlight the important properties and requirements of stream-aware scheduling algorithms. We then discuss how to handle variable latency and capacity in Section V.

On top of this network model, the scheduler can continuously transmit data on a subset of paths and can change its path allocation at any time. We also assume that the scheduler has access to the total size $S$ of the message, for instance through a HTTP/2 header, and that application data is always available when the scheduler needs it.

We compute the minimum completion time as a function of $(D_p)$, $(R_p)$ and the message size $S$, assuming that paths are ordered by increasing latency. The key finding is that, for two paths, there is a size threshold $S_{lim}$: below this threshold, the completion time is minimized by taking only the first path, because the latency of the second path is larger than the total completion time. Above this threshold, both paths are useful, until the remaining size crosses the threshold: at this point, all remaining data should be sent on the first path. This allows both paths to complete their transfer at the same time from the point of view of the receiver. The completion time as a function of the message size $S$ is given by:

$$C^*_{S \leq S_{lim}} = D_1 + \frac{S}{R_1} \tag{1}$$

$$C^*_{S \geq S_{lim}} = D_2 + \frac{S - S_{lim}}{R_1 + R_2} \tag{2}$$

$$\text{where} \qquad S_{lim} = R_1 \cdot (D_2 - D_1) \tag{3}$$

These equations are actually valid at any point in time, giving the *remaining completion time* as a function of the *remaining size*. This *stability* property yields a natural scheduling

algorithm that, at any time, simply compares the remaining size with the threshold $S_{lim}$. Fig. 3 shows such an ECF schedule with two paths. At first, both paths are used. Starting from time $t = 12.86$, the remaining size drops below the $S_{lim}$ threshold, so only the path with lowest latency is used.

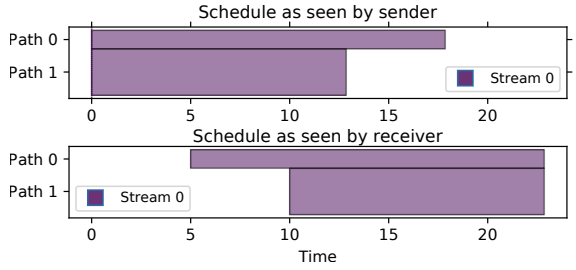


Fig. 3. Example of an optimal schedule computed with ECF for a single stream, as seen from the sender (top) and the receiver (bottom). Notice how data on the two paths completes simultaneously on the receiver side. Path 0 has rate 200 KB/s and latency 5 ms, while path 1 has rate 500 KB/s and latency 10 ms. The stream has a size of 10 KB.

For $n$ paths, we can show recursively that there exists $n-1$ similar size thresholds: as the stream size increases, it becomes more and more useful to use paths with larger latency. An example curve with 3 paths is shown in Fig. 4.

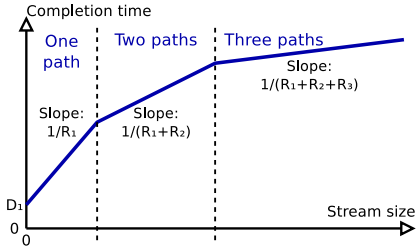


Fig. 4. Completion time as a function of stream size using ECF. Here, $D_1$ is the delay of the lowest-delay path, while $R_i$ is the rate (in bit/s) of path $i$ where paths are ordered by increasing delay.

## III. SRPT-ECF: Optimal Stream-Aware Multipath Scheduling

We now consider multiple streams, each transmitting an atomic *message* $(m_k)$ of size $S_k$ such as a web resource. We assume that all messages are created at the same time and are available at the sender. We use the same network model as in Section II and our goal is to minimize the completion time of individual messages.

### A. The SRPT-ECF scheduling algorithm

Our algorithm works in two steps:

*a) order messages:* order messages according to the *Shortest Isolated Remaining Completion Time* principle: compute the remaining completion time for each message *if it was alone* using ECF, and schedule messages with smaller such *isolated* completion time first. This principle is inspired from the SRPT algorithm (Shortest Remaining Processing Time) from classical scheduling theory, hence the name of our algorithm. With our hypotheses, it boils down to scheduling *smaller messages first* because the function from Fig. 4 is non-decreasing; in a real implementation, we would need to keep track of in-flight packets to estimate the isolated completion time of each stream.

*b) allocate path resources:* for each stream in order, schedule it on the available paths using Earliest Completion First (ECF). For a given stream, the size we consider to determine which paths to use (see Fig. 4) accounts for all the streams that are scheduled before. Thus, the effective size of message $m_k$ for ECF is: $S_k^{ECF} = \sum_{i \leq k} S_i$. This is because we schedule the data of all messages $(m_i)_{i<k}$ before scheduling message $m_k$.

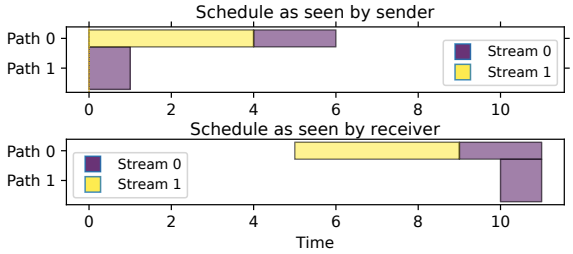


Fig. 5. Optimal schedule computed with SRPT-ECF for two streams with the same paths as in Fig. 3. Stream 0 has size 900 B while Stream 1 has size 800 B. Notice how the smaller stream gets priority but only uses the shortest-latency path thanks to ECF, while the bigger stream exploits the unused resources on Path 1. However, thanks again to ECF, it stops using Path 1 early to ensure simultaneous completion on both paths. The sequence of completion times is $(9, 11)$ and is optimal.

Our SRPT-ECF algorithm has desirable properties: it prioritizes smaller streams, lowering their completion time without having a significant impact on larger streams. In addition, since ECF may not use all paths to transmit a small message, unused resources can be used by larger and thus lower-priority streams. This behavior is illustrated in Fig. 5 where Stream 0 is scheduled with ECF with an effective size of $S_0^{ECF} = S_1 + S_0$, causing it to use the high-latency path while it waits for access to the low-latency path.

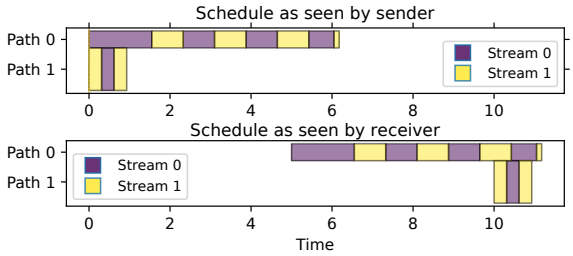


Fig. 6. Same situation as Fig. 5, but with the SA-ECF scheduler [8] (Weighted Round-Robin + ECF) with 150 bytes of quantum and equal weights. The overall completion time is roughly the same as SRPT-ECF, but all streams have the worst possible completion time among ECF schedulers: the sequence of completion times is $(11, 11.2)$.

We compare our algorithm with Round-Robin (SA-ECF [8]) using the simulator from Section IV-C. The result is shown in Fig. 6: Stream 0 has roughly the same completion time in both cases, but Round-Robin increases the completion of time of Stream 1. Overall, Round-Robin is a net loss: by introducing

more fairness, it actually made the situation worse for a stream while not improving the situation for the other stream.

### B. SRPT-ECF achieves an optimal completion sequence

Let $(C_i)$ be the ordered sequence of completion times achieved by SRPT-ECF, and $(C'_i)$ the ordered sequence of completion times achieved by any other scheduling algorithm. The order in which messages complete may not be the same for both cases: $C_1$ might be the completion time of message $m_1$ while $C'_1$ might be the completion time of message $m_3$.

The sequence $(C_i)$ is optimal in the following sense:

$$\forall i, \quad C_i \leq C'_i \tag{4}$$

This result can be proved by showing that any optimal schedule is *work-conserving*, *i.e.* the overall schedule behaves like ECF with a single stream of size $\sum S_k$, and that the SRPT ordering is the best ordering. This is because the total completion time $C^*(\sum_{k\in\mathcal{S}} \mathrm{S}_k)$ of any subset of streams $\mathcal{S}$ is a non-decreasing function of their total size.

### C. Properties and extensions

*a) Optimal average completion time:* In particular, our optimality result implies that SRPT-ECF is also optimal for the average completion time of streams.

*b) Stability:* Similarly to Section II, the SRPT-ECF algorithm is stable: the order in which messages are considered will stay the same over time. This is because the Isolated Remaining Completion Time of each stream will reduce faster for higher-priority streams: therefore, their order is not modified. This stability property means that streams have *no regrets* about their past actions, and avoids *priority inversions* that would be harmful for stream completion times.

*c) Application-provided priorities:* Our algorithm can easily accommodate priorities provided by the application: in the first step, we simply order lexicographically on the couples (priority, isolated completion time), and we apply our unchanged ECF strategy in the second step. Thus, this is a *strict priority* scheme, as opposed to a Weight Round Robin mechanism. But we keep the optimal completion pattern of SRPT-ECF, and lower-priority streams can use resources that are not used by higher-priority streams.

## IV. Online multipath scheduling

In Section III, we assumed that all streams were available at the same time. In practice, however, the application may decide to create and transmit a new message at any time: this becomes an *online scheduling problem* where each message $m$ is *released* at a given date $r_m$. The scheduler needs to take decision without knowing future release dates, potentially leading to poor decisions in hindsight. In this context, the *completion time* of a message is defined relatively to its release date, i.e. the completion time equals $C_m = f_m - r_m$ where $f_m$ is the *finish date* at which the message is completely received.

Our SRPT-ECF algorithm can still be used, but it is no longer optimal: an offline algorithm that knows about future messages could anticipate and produce a slightly better schedule. Still, we show that SRPT-ECF remains desirable.

### A. Online SRPT-ECF algorithm

Our algorithm still works exactly as described in Section III-A, with the difference that the *Isolated Remaining Completion Time* can become a bit more complicated to compute.

Assume that we already have a set of messages $(m_k)_{1\leq k\leq n-1}$ already being scheduled and transmitted with SRPT-ECF. Then, at time $r_n$, a new message $m_n$ is released and wants to be scheduled: the current schedule may need to be changed to satisfy the requirements of SRPT. We need to recompute and compare the *Isolated Remaining Completion Time* for each message, that is: assuming we dedicate all paths resources to a message, how much time would it need to complete? A key difference with Section III-A is that part of a message may already be in-flight, possibly impacting its remaining completion time. This is especially the case when a packet is in-flight on a high-latency path: even if all future packets use a low-latency path, we still need to wait for this in-flight packet to arrive. In practice, the Isolated Remaining Completion Time of a message $m_k$ with size $S_k$ is the maximum of two terms: the optimal completion time given by ECF (which only depends on $S_k$) and the time $T_{\text{in-flight}}$ needed for in-flight packets to reach the destination:

$$C_{\text{isolated}}(m_k) = \max\Big[C^*_{\text{ECF}}(S_k), T_{\text{in-flight}}(m_k)\Big] \tag{5}$$

This second term can be computed by predicting the date at which in-flight packets will finish on all paths. A simple way to do this is to record the last time at which we sent a packet on a path $p$ and add the path delay $D_p$, assuming it does not change over time, and take the maximum over all paths. Once the Isolated Remaining Completion Time is computed for all messages, we order messages from smaller to bigger and use ECF to allocate paths to messages, as before.

Thanks to the stability property, we don't need to recompute $C_{\text{isolated}}$ for all messages, since the relative order of existing messages is unchanged: the new message can simply inserted in the ordered list of messages, which can be done with only $O(log(n))$ operations.

Fig. 7 shows an online example of SRPT-ECF where a big stream is pre-empted by two small streams in succession.

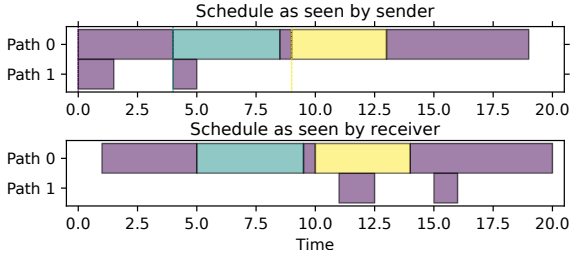


Fig. 7. Example of SRPT-ECF running online. Stream 0 is initially alone and uses ECF as in Fig. 3. When Stream 1 arrives ($t = 4$), Stream 0 gets preempted immediately: as a result, it starts using Path 1 again so that data on both paths finishes simultaneously, accounting for Stream 1. When Stream 2 arrives ($t = 9$), Stream 0 is preempted again but this time it is too late to use Path 1, so it just waits for Stream 2 to finish.

### B. Comparison with offline algorithms

To understand how much SRPT-ECF can lose by not anticipating future events, we compare it to *offline scheduling algorithms* that know about future stream releases and are able to anticipate. Our goal is to illustrate the behavior of SRPT-ECF, rather than conducting a thorough analysis of offline-to-online performance ratio.

We implement a simple offline algorithm that tests all possible priority orderings between streams. For each priority ordering, it allocates resources to streams in this order while ensuring simultaneous completion on all paths, giving resources to a lower-priority stream if a higher-priority stream has not yet been released. The algorithm then selects the ordering that yields the best value of the chosen metric, which could be for instance the average completion time or the maximum completion time.

In Fig. 8 we show the result of this offline algorithm in the same setting as Fig. 7, taking the average completion time as metric to minimize. For Stream 0, the offline algorithm does 18% better than SRPT-ECF. This example illustrates that SRPT-ECF works best when messages are available in batches; in addition, the application should inform the scheduler as soon as possible about data it plans to send, even if such data is not yet available.

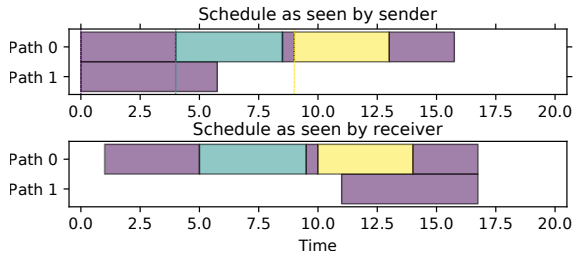


Fig. 8. Optimal schedule obtained by an offline algorithm that minimizes the average completion time. Stream 0 can anticipate the arrivals of Streams 1 and 2, so it uses Path 1 for a longer time to ensure both paths finish at the same time.

### C. Evaluation

To evaluate our SRPT-ECF algorithm against other scheduling algorithms, we use trace-based simulation. We produce traces with an actual web browser that loads a web page with HTTP/2 on a single path, using `webpagetest.org`. The tool emulates a low-end residential connection (DSL with 1.5 Mbps download capacity and 50 ms of additional RTT). For each web resource, we record the date of the request and the size of the response. We then select a single connection from this trace and replay it in a simulator[1] implementing our multipath model: for each web resource, we create a message of the same size in a multipath connection from the server to the client; the release date $r_m$ is the date of the original request. We use two paths with comparable combined capacity and end-to-end latency compared to the original setup: a first path with 80% of the capacity and 80% of the end-to-end latency (DSL), and a second path with 40% of the capacity and 180% of the end-to-end latency (mobile). Note that we don't compare our results with the original trace because it would be meaningless: we simply compare different schedulers on the same trace replay.

We take several steps to ensure the trace is meaningful: 1) we ensure that data is cached on the server by repeating the experiment and keeping the last run: we can thus assume that the server can send the content as soon as it receives the client request; 2) when recording the trace, we emulate a slow DSL connection to make sure the page load is network-bound and not CPU-bound, and also to have many resources loading simultaneously; 3) when replaying the trace, we simulate a slightly faster overall network (120% of the original capacity) to respect dependencies without having to enforce them explicitly. Indeed, each web page has an implicit dependency graph [13], but instead of trying to determine this graph, our method makes sure that any prerequisite requests are finished by the time a new request is made. That is, any dependency in the original web page produces two independent groups of requests in the replay.

Fig. 9 shows the CDF of stream completion times with SRPT-ECF and SA-ECF [8] when loading the 132 images of a wikipedia page. The Weighted Round-Robin component of SA-ECF is implemented with the same weight for all streams. We also include two simpler schedulers based on MinRTT: First-Come-First-Serve (FCFS-minRTT) that schedules streams in their order of arrival, and SRPT-minRTT that orders streams by size. There are two main results: 1) ECF exploits the low-latency path to reduce completion time for small streams, while MinRTT does not; 2) SRPT brings a very significant improvement for the tail of resources compared to both Round-Robin (SA-ECF) and FCFS.

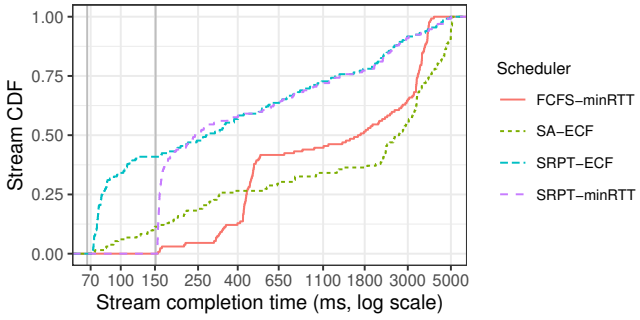


Fig. 9. Simulation of a wikipedia page load by replaying a trace (ID `200413_FH_478b18e178c0fbf2ec9312686630e510`, run 3, connection 2). Path 0: 1050 Kbit/s, 67 ms. Path 1: 750 Kbit/s, 151 ms. Path latency is indicated with the vertical bars. SRPT-ECF exhibits low completion times, thanks to its combination of ECF (left part of the CDF) and SRPT (tail of the CDF).

To better understand this second effect, we plot the number of active streams at each instant in Fig.10. SRPT-based schedulers reduce the number of active streams as much as possible by prioritizing small streams, thereby completing them sooner.

[1]Source code is available at https://gricad-gitlab.univ-grenoble-alpes.fr/jonglezb/multipathsim/-/tree/master/model

With FCFS, small streams suffer when they have to wait behind a larger stream, increasing both the backlog of active streams and the completion time. SA-ECF accumulates even more work because of Round-Robin, but smaller streams do not have to wait for large streams to finish, yielding better completion time for small streams compared to FCFS. Overall, combining SRPT and ECF gives the best results.

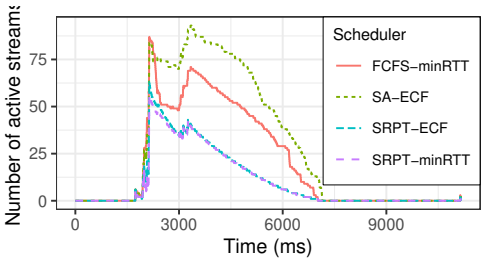


Fig. 10. Scheduler occupation over time during the trace replay. Streams are created in three visible bursts. The schedulers have different strategies for handling the backlog of active streams.

## V. Practical considerations

All results so far are based on the simple network model from Section II. In real networks, the latency and available capacity can vary significantly over time, and this can have a large impact on the effectiveness of scheduling algorithms. We sketch some ways in which our SRPT-ECF could be adapted to cope with this issue.

The main problem is the uncertainty associated with measured latency and capacity: past measurements may not be good predictors of future network conditions. In addition, it takes one RTT for the sender to become aware of a significant change in the network. The key challenge is to anticipate these unexpected changes while keeping the associated cost low. The DEMS scheduler [4] uses several interesting techniques that could be applied to an implementation of SRPT-ECF: using the one-way delay difference between paths instead of the RTT, controlled data duplication at the end of a stream, adaptation based on the variance of latency samples. We leave a more detailed development on implementation aspects of SRPT-ECF for future work.

## VI. Conclusion

Round-Robin is often used when several objects need to share resources, because it avoids starvation. However, when looking at completion time, Round-Robin hurts smaller flows disproportionally without actually improving the completion time of other flows.

More generally, Round-Robin would make sense for concurrent flows, where it would be unfair to give all resources in turn to each flow, or in case of infinite flows. But in a situation where all flows belong to the same user and have a finite size, fairness is not relevant. When flows cooperate with each other, each flow can tolerate some unfairness to allow other flows to achieve better results. Of course, the application can still define priorities to indicate which flow needs the resources the most.

Despite its apparent simplicity, our SRPT-ECF algorithm implements this idea by achieving an optimal pattern of completion times. This brings highly desirable properties such as allowing very different types of streams to coexist peacefully: small or important messages enjoy strict priority to use the lowest-latency paths, while bulk transfers can still use other higher-latency paths. This has the additional advantage of lowering the risk of starving big flows, because they can exploit unused resources offered by the multipath setup.

We expect our algorithm to help MPQUIC improve HTTP page load time, opening the door to efficient implementations of Multipath HTTP/3. Future work will be focused on implementing our algorithm in a MPQUIC scheduler.

## References


[1] Q. De Coninck and O. Bonaventure. Multipath QUIC: Design and Evaluation. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, pages 160–166, New York, NY, USA, 2017. ACM.

[2] T. Dreibholz, R. Seggelmann, M. Tüxen, and E. P. Rathgeb. Transmission scheduling optimizations for concurrent multipath transfer. In *Proceedings of the 8th International Workshop on Protocols for Future, Large-Scale and Diverse Network Transports (PFLDNeT)*, volume 8, 2010.

[3] S. Ferlin, O. Alay, O. Mehani, and R. Boreli. BLEST: Blocking estimation-based MPTCP scheduler for heterogeneous networks. In *2016 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 431–439, May 2016.

[4] Y. E. Guo, A. Nikravesh, Z. M. Mao, F. Qian, and S. Sen. Accelerating Multipath Transport Through Balanced Subflow Completion. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom '17, pages 141–153, Snowbird, Utah, USA, Oct. 2017. Association for Computing Machinery.

[5] A. Langley, A. Riddoch, et al. The QUIC transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 183–196, New York, NY, USA, 2017. ACM.

[6] Y.-s. Lim, E. M. Nahum, D. Towsley, and R. J. Gibbens. ECF: An MPTCP Path Scheduler to Manage Heterogeneous Paths. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, pages 147–159, New York, NY, USA, 2017. ACM.

[7] C. Paasch, S. Ferlin, O. Alay, and O. Bonaventure. Experimental Evaluation of Multipath TCP Schedulers. In *Proceedings of the 2014 ACM SIGCOMM Workshop on Capacity Sharing Workshop*, CSWS '14, pages 27–32, New York, NY, USA, 2014. ACM.

[8] A. Rabitsch, P. Hurtig, and A. Brunstrom. A Stream-Aware Multipath QUIC Scheduler for Heterogeneous Paths. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, EPIQ'18, pages 29–35, New York, NY, USA, 2018. ACM.

[9] H. Shi, Y. Cui, X. Wang, Y. Hu, M. Dai, F. Wang, and K. Zheng. STMS: Improving MPTCP throughput under heterogeneous networks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 719–730, Boston, MA, July 2018. USENIX Association.

[10] X. Shi, L. Wang, F. Zhang, and Z. Liu. FStream: Flexible Stream Scheduling and Prioritizing in Multipath-QUIC. In *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 921–924, Dec. 2019. ISSN: 1521-9097.

[11] T. Viernickel, A. Froemmgen, A. Rizk, B. Koldehofe, and R. Steinmetz. Multipath QUIC: A Deployable Multipath Transport Protocol. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–7, May 2018.

[12] J. Wang, Y. Gao, and C. Xu. A Multipath QUIC Scheduler for Mobile HTTP/2. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, APNet '19, pages 43–49, Beijing, China, Aug. 2019. Association for Computing Machinery.

[13] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying Page Load Performance with WProf. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 473–485, Lombard, IL, 2013. USENIX.