Chair of Network Architectures and Services
School of Computation, Information, and Technology
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

SCHOOL OF COMPUTATION, INFORMATION, AND TECHNOLOGY

INFORMATICS

MASTER'S THESIS IN INFORMATICS

State-of-the-art Multipath Scheduler for QUIC

Daniel Petri Rocha

Technical University of Munich

School of Computation, Information, and Technology

Informatics

Master's Thesis in Informatics

State-of-the-art Multipath Scheduler for QUIC

Moderner Multipfad-Scheduler für QUIC

Author:	Daniel Petri Rocha
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Kilian Holzinger, Marcel Kempf
Date:	August 15, 2024

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Garching, August 15, 2024

Location, Date

Signature

Abstract

Multipath QUIC (MPQUIC) aggregates the bandwidth of multiple network paths. The protocol extension, however, does not specify a general-purpose scheduler tailored to QUIC's stream multiplexing capabilities. As a result, current MPQUIC implementations rely on generic scheduling strategies explicitly designed for Multipath TCP without accounting for the differences between the protocols. Given that recent scheduling theory advancements have enabled algorithms for hierarchical max-min fairness at deployable rates, we integrate such an approach into a fork of Cloudflare's QUIC library *quiche* featuring multipath support. Applications can classfully divide traffic per connection by specifying a weighted hierarchy with minimal coupling to the transport layer. Traffic classes are isolated from each other and receive strategy-proof minimum rate guarantees; unused bandwidth is fairly re-distributed to classes with unsatisfied requests. We show that these properties are application protocol-agnostic, being suitable for HTTP/3. Our scheduler schedules streams with byte granularity and is compatible with modern multipath scheduling strategies that optimize stream completion times and out-of-order packets. For web-like traffic, it handles path heterogeneity better than quiche's default scheduler and has lower stream completion times.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

People's smartphones can automatically switch between Wi-Fi and cellular data when entering or leaving a place. Although the user changes from one network to another, they continue to be reachable or capable of reaching others, showing that many paths over the Internet lead to the same destination. Each path has different capabilities. The Internet encountered on the go, such as on a train or airplane, is notably slow and unreliable. Choosing *which* to use is crucial for the user's experience if multiple paths exist. Furthermore, *how* they are used matters, too: a video call should not drop just because large photos are being downloaded in the background by the same application, for instance.

A proposed feature for the general-purpose transport protocol QUIC [1] is enabling the use of multiple paths of a network simultaneously [2]. To the application layer, a Multipath QUIC (MPQUIC) extension appears as a single logical connection [3] whose bandwidth is the aggregate of the available paths. From the QUIC library's perspective, paths are 4-tuples containing the source and destination Internet Protocol (IP) addresses and ports [1]. In practice, an IP address is assigned to an interface connected to a network. These are increasingly dual-stack, supporting both IPv4 and IPv6 [4]. MPQUIC is enabled by multihomed devices whose interfaces are attached to distinct networks simultaneously, such as cellular and WLAN.

MPQUIC aids service quality and experience in various contexts. The throughput is higher, and in real-time applications, multipath redundancy can reduce unacceptable tail latencies with proactive loss recovery mechanisms [3]. While singlepath QUIC handles connection migration out of the box when, e.g., a mobile endpoint switches to a new network, MPQUIC can make this handover even smoother [5].

The MPQUIC specification proposed by the Internet Engineering Task Force (IETF) QUIC Working Group [2] provides little engineering guidance regarding a packet scheduler, solely basing the choice of which path to use on its smoothed round-trip-time (RTT) value and variance. None is given for a stream scheduler. With the protocol stack of the third major version of the HyperText Transfer Protocol (HTTP) building atop QUIC [6], theorized or deployed schedulers are often tightly coupled to the web use case. But QUIC, built on top of the User Datagram Protocol (UDP), also finds applications in cloud gaming, media streaming, teleoperated driving, and remote surgery [5], for example. Thus, the implementation of a scheduler should remain application protocol-agnostic. The standardized analog extension for the Transmission Control Protocol (TCP), Multipath TCP (MPTCP), stirred experimental research into the design of high-end-system throughput schedulers [7] that provided valuable lessons and concrete implementations. Nonetheless, as seen in some libraries, their reuse for MPQUIC fails to leverage its unique properties [8].

Therefore, this thesis addresses the imperative to design, implement, and evaluate a fair and flexible MPQUIC scheduler. To that end, we port the Hierarchical Link Sharing (HLS) Queueing Discipline (QDisc) to QUIC. Modifications are needed as it was designed initially for packets, not streams. Its advantages lie in the fine-grained control it provides to an application, which specifies a weighted hierarchy of traffic classes stored at the connection level. Our transport-layer scheduler then uses the hierarchy to schedule data with byte granularity following a hierarchical max-min fair allocation. Traffic classes are guaranteed minimum rates and fully isolated from one another. Then, we integrate it with the Stream-Aware Earliest Completion First (SA-ECF) stream- and path scheduler. Unlike non-stream-aware schedulers, it can estimate on a per-stream basis whether a given path will delay the stream's completion. When that is the case, the stream must wait for a faster path to re-open, handling path heterogeneity sensibly.

Chapter 2 explains the scheduling and fairness mechanisms behind existing approaches essential to this thesis. Chapter 3 focuses on the literature that tackles similar scheduling challenges and how our developed scheduler positions itself in that context, with Chapter 4 presenting our solution approach. Chapter 5 describes its implementation in an MPQUIC fork [9] of the open-source IETF QUIC library quiche. Experiments done and evaluated on a testbed are presented in Chapter 6. We conclude this thesis in Chapter 7 by providing an outlook for future work in the context of a final discussion of our results.

Chapter 2

Background

In this chapter, we provide the necessary background to understand the scheduler currently implemented in the QUIC library quiche, which we subsequently replace in the development phase. We start with an introduction to general scheduling strategies, which, despite simple, are the foundation of complex packet schedulers such as the HLS QDisc. As it is Hierarchical Max-Min (HMM)-fair, we provide illustrative examples for this central concept in Section 2.3 to motivate the topic as a short crash course.

2.1 Generic Schedulers

Abstract strategies are a good starting point to provide intuition behind a scheduler's operation, as general-purpose schedulers are portable and straightforward to implement and can later be tailored to application-specific needs.

2.1.1 Round-Robin

A Round-Robin (RR) scheduling discipline ensures all paths, packets, or streams are evenly visited over time by cyclically iterating over a list of available schedulee identifiers (IDs). These may be assigned different priorities so that some scheduling subjects are favored at the expense of others. Priority-based network resource allocation over streams can be achieved with a Weighted Round-Robin (WRR). Each stream receives the opportunity to emit a number of packets equal to its weight when it is visited in a scheduling round [5]. Since these visits occur regularly, no stream can monopolize the transmission and starve others.

A RR path scheduler performs well in ideal network conditions because it combines the bandwidth of both paths [5], maximizing their utilization. In a WRR, packets are non-

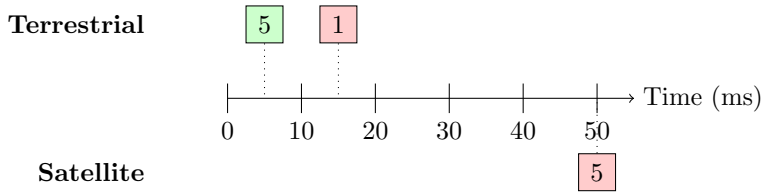

Figure 2.1: LowRTT-scheduled multipath packet arrival

uniformly distributed. The available path list being iterated on has several duplicated entries for higher-importance paths, with its ordering defining whether sending occurs in a bursty or interleaved fashion [4]. Due to potentially high latency differences among paths, however, Jonglez et al. [8] find that "serving streams using a Round-Robin strategy yields poor performance when looking at stream completion time" in web use-case contexts.

2.1.2 Lowest-RTT-First

The Lowest-RTT-First (LowRTT) scheduler sends on the path whose smoothed RTT is smallest, filling its congestion window before moving on to the next fastest one. The smoothed value is a running average of RTT measurements over time, which may give recent samples more weight. The scheduler clashes with the unfortunate reality of dynamically changing conditions found in heterogeneous mobile wireless networks, as it is intended for homogeneous paths. Depending on how heterogeneous the paths are, a markedly inferior second path can severely degrade LowRTT's performance due to out-of-order (OFO) packets.

Consider a scenario [10] where a terrestrial link has an RTT of 10 ms, while a satellite route exhibits an RTT of 100 ms. In both cases, their congestion windows support the transmission of five packets, and the one-way delays are symmetrical. Figure 2.1 illustrates when, on which path, and how many packets arrive at the peer in this example. A flow of 11 packets would see the first five packets scheduled on the fastest path, followed by a further five on the slowest. 10 ms later, the sender receives an acknowledgment on the terrestrial link, which opens its congestion window anew for the last packet. It is delivered after 15 ms while the second packet burst is still in flight, causing intra-stream Head-of-Line Blocking (HoLB) due to OFO arrivals in red on top of the long wait time. The fast path quickly became idle even though the second path still transfers data.

Although it is known that LowRTT can lead to these circumstances where the fast path is underutilized, it is MPTCP's default scheduler and found in the Linux kernel implementation. It was naively ported to MPQUIC in quiche without addressing the

fact that MPTCP's use case is fundamentally different to MPQUIC's, as it splits a single stream over several sub-flows instead of multiplexing streams over many paths [11] [8].

2.1.3 Strict Priority

A strict priority path scheduler chooses the first available path from a priority-sorted list. Paths are checked in order of descending priority; lower-priority paths are only considered if all higher-priority paths are unavailable due to the used congestion control scheme [4]. As with RR, the same logic applies to jobs, tasks, or packets that desire scheduling; quiche uses it for streams.

2.2 Prioritization Schemes for HTTP

Quiche schedules streams based on a strict priority scheme. Internally, the urgency `u` ranges from 0 to 255. The default priority is 127, with lower values transmitted on the wire first [12]. Alongside the urgency, a boolean flag `i` indicates whether the stream is incremental, meaning that it can be processed online.

Streams are first sorted based on their urgency. At each urgency level, non-incremental requests are sent out in full, ordered by their stream IDs. Then, the incremental streams are subordinate to a round-robin that emits a packet for each partial request. The intuition behind the scheduler's strategy is that it allocates more bandwidth to important streams while letting secondary streams that can be progressively handled divide the rest amongst them [13].

This algorithm is compatible with the scheduling recommendations outlined by the Extensible Prioritization Scheme for HTTP [14] and is "an important piece of deploying HTTP/3." [15]. The Request for Comments (RFC) retains the incremental flag but narrows the urgency's range down to 0–7 for simplicity. The proposed standard also defines how priority signals are communicated, which the original HTTP/3 specification did not provide a method for [6]. It aims to minimize the user-centric Largest Contentful Paint (LCP) loading experience metric, which measures the time it takes for the largest visible image or text block to render [16].

The Extensible Prioritization Scheme replaces stream priorities from HTTP/2. Using a parameter in a `SETTINGS` frame, endpoints can state to their peers that HTTP/2 priorities are disabled [6]. These were deprecated in later protocol revisions as they were not widely adopted, but backward compatibility was kept [14]. HTTP/2 priorities are weighted edges in an unbalanced dependency tree where each stream is a vertex. The explicit dependencies signify that the parent stream needs to be done sending before

the child can start doing so. The relative — not absolute — weights range from 1 to 256, representing the proportion in which resources are allocated. Each level of the dependency tree independently handles prioritization.

Figure 2.2 shows the dependency tree of a bare-bones web page. In that example, every stream depends on the HyperText Markup Language (HTML) document to be displayed or function. The JavaScript (JS) code needs the Cascading Style Sheets (CSS) file, so it does not profit from arriving before the style sheet. The CSS file receives 64% of the bandwidth distributed by the root, as it splits the available capacity with its siblings in a ratio of 256:72:72. As a result, images outside of the viewport obtain a smaller share of the remaining bandwidth.

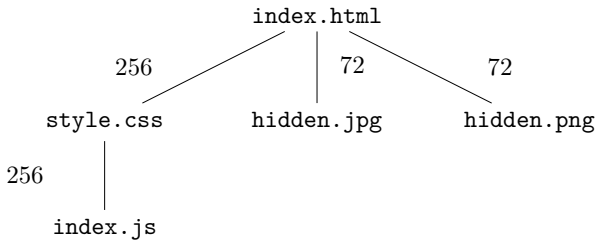

Figure 2.2: HTTP/2 dependency tree

It stands to reason that due to modern web pages consisting of hundreds of separate requests, such a structure can rapidly become difficult to reason about as a web developer. The resulting lack of deployment and interoperability issues led to little adoption of HTTP/2's dependency trees. Nonetheless, they remain a robust model that several MPQUIC schedulers from Section 3.1 incorporated into their design.

With HTTP/3 priorities, similar results can be obtained with less cognitive overhead. Suppose a client requests `index.html` from an upstream server, signaling a preference of `u=0, i=true`, since it can be parsed as it arrives. Since the CSS file should arrive before the JS script, we respectively assign them `u=1, i=false` and `u=2, i=false`. Any further off-screen element is given `u=3, i=true`.

Figure 2.3 illustrates the order in which the requests are fulfilled. The stream with the HTML payload is emitted first after ordering streams based on urgency, followed by the CSS file and then the JS script. The tie at the third urgency level is handled with a packet-by-packet RR, as both streams are marked as incremental. Had they been flagged as non-incremental, their IDs would have been used to determine the precedence instead.

In Subsection 6.1.4, we run this example with quiche to have a baseline against which to compare our Fair Stream Scheduler.

Figure 2.3: Theoretical HTTP/3 scheduling result

2.3 Fairness Fundamentals

In this section, we introduce max-min fairness directly with its weighted counterpart, since a max-min fair allocation simply has its weights set to one. Similarly, we later see that a weighted max-min allocation is a subset of the more general HMM-fair allocation.

Weighted Max-Min Fairness

Let us instantiate a practical example in the context of a stream scheduler. Bytes are the limited resource competed for in a scheduling round. The scheduler, i.e., the root node, can at most distribute its capacity C per round. In Figure 2.4, 1000 B are divided amongst the traffic classes X, Y, and Z, that together form the set of classes $\mathcal{N}$.

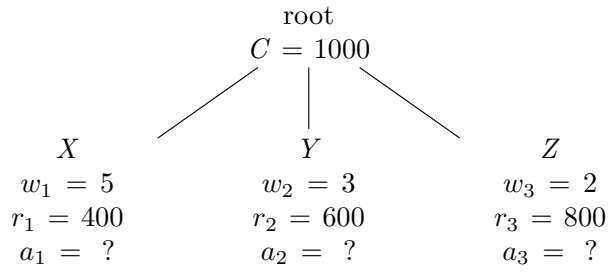

Figure 2.4: Initial state of a weighted max-min fair allocation

A traffic class i is associated with a request r_i, weight w_i, and an allocation $a_i \leq r_i$ yet to be determined [17]. The weights indicate that they should split the available capacity in a ratio of 5:3:2. The difficulty in allocating the requests lies in the sum of the requests in Figure 2.4 being 1800 B, i.e., higher than the capacity of 1000 B. The implication is that some requests cannot be fulfilled, so we look for an allocation that maximizes the minimum amount of data that unsatisfied streams receive. If the overall demand were smaller than the capacity, there would be no need to apply max-min fairness.

The algorithm begins by obtaining an initial fair share f that is the quotient of the capacity and the total requesting sources [18]. Formula 2.1 presents the mathematical expression for this first iteration.

$$f = \frac{C}{\sum_i w_i} \tag{2.1}$$

The fair share value states how many bytes a stream with a weight of one can receive. In our example, that equals $^{1000}/_{10} = 100$ bytes. Taking each stream's weight into account yields Formula 2.2 for the allocation.

$$a_i = \min\{r_i, w_i f\} \tag{2.2}$$

Class X can, therefore, get up to 500 bytes. Since it only requests 400, 100 remain that can be redistributed fairly to Y and Z in a second iteration as their requests have not been fulfilled by allocations of 300 and 200, respectively. Figure 2.5 shows the state of the process before an additional round starts distributing the remaining capacity. Satisfied classes are represented in green; the unsatisfied in red.

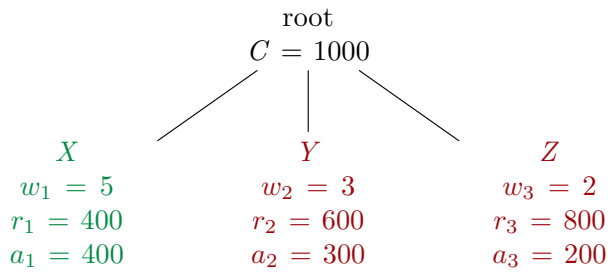

Figure 2.5: State of a weighted max-min fair allocation after the first iteration

When starting one or more *surplus* rounds, the process is repeated considering unsatisfied classes until the capacity is fully depleted or all classes are satisfied [17]. The sum of the remaining weights is 5 for a new capacity of 100. Due to the new fair share of 20, Y obtains 60 more and Z 40. With the resource exhausted, Figure 2.6 shows the final result.

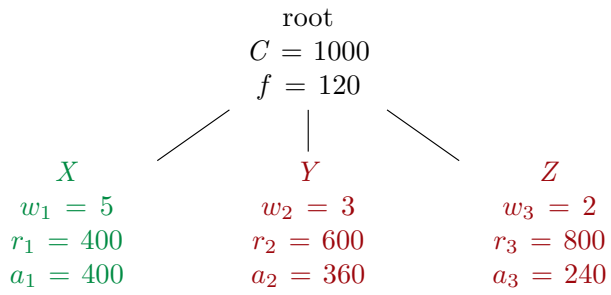

Figure 2.6: Final state of a weighted max-min fair allocation

Intuitively, the algorithm computes the allocation for each class, satisfies classes requesting less than their allocation, and then repeats the process to equally split the surplus among the remaining unsatisfied classes. Inspecting the final result reveals that the

root essentially distributed a final fair share of 120 to everyone; class X just happened to need less. Therefore, it does not feel taken advantage of. Although classes Y and Z remain unsatisfied, they received the resource in the same proportion, i.e., a multiple of the fair share according to their weights. Furthermore, the result is strategy-proof because Y and Z cannot increase their balance by gaming the system: increasing their requests further does not alter the allocation. Mathematically, properties 2.3 and 2.4 hold [19].

$$a_i < r_i \Rightarrow \frac{a_i}{w_i} \geq \frac{a_j}{w_j}, \forall j \in \mathcal{N} \tag{2.3}$$

$$\sum_{j \in \mathcal{N}} a_j = \min(\sum_{j \in \mathcal{N}} r_j, C) \tag{2.4}$$

Hierarchical Max-Min Fairness

Even though the previous example was put forward as a non-hierarchical, weighted max-min fair allocation, the root can be interpreted as a full-fledged class in a flat hierarchy. We add a new level to the hierarchy by giving children to the classes X and Z in Figure 2.7.

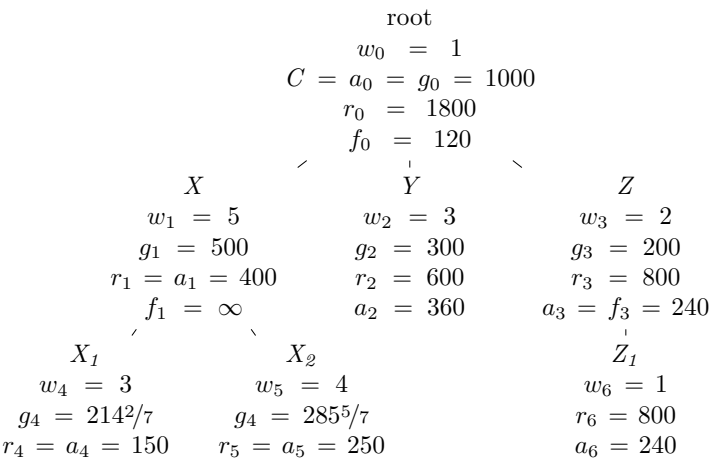

Figure 2.7: Output of a hierarchical, weighted max-min fair allocation

$\mathcal{N}$ then is the union of the root, the leaves $\mathcal{L}$, and the remaining *internal classes* $\mathcal{I}$ [19]. Operations supported by a hierarchy are presented in Table 2.1 [17] alongside the explanation of key terms.

With this formal notation, non-hierarchical max-min fairness can be generalized as a special case of HMM. As visualized in Figure 2.7, each parent distributes its available

Term	Description	Example
root	Class at the hierarchy's top	The node with $C = 1000\,\mathrm{B}$
$\mathcal{L}$	Classes at the hierarchy's bottom	$\{X_1, X_2, Y, Z_1\}$
$\mathcal{I}$	Internal classes, neither leaves nor root	$\{X, Z\}$
$\mathcal{N}$	All classes: $\mathcal{L} \cup \mathcal{I} \cup \{\text{root}\}$	$\{\text{root}, X, Y, Z, X_1, X_2, Z_1\}$
$p(i)$	Parent of node i	$p(X_1) = X$
$child(i)$	Children of node i	$child(X) = \{X_1, X_2\}$
$sib(i)$	Siblings of node i, including itself	$sib(X) = \{X, Y, Z\}$
$anc(i)$	Ancestors of i	$anc(Z_1) = \{Z, \text{root}\}$
$desc(i)$	Classes with i as ancestor	$desc(\text{root}) = \mathcal{L} \cup \mathcal{I}$
$ldesc(i)$	Leaf descendants with i as ancestor	$ldesc(\text{root}) = \mathcal{L}$

Table 2.1: Terminology for HMM fairness

allocation to their children using the previously established process. Hence, each parent, i.e., the root nodes of the sub-hierarchies, computes a fair share. It is infinite when all requests can be satisfied. Formulas 2.5 state the adapted computation for the request and allocation values [19].

$$r_i = \sum_{j \in child(i)} r_j, \quad a_i = \sum_{j \in child(i)} a_j, \ \forall i \in \mathcal{I} \cup \{\text{root}\} \tag{2.5}$$

As before, the first rule in 2.6 ensures that unsatisfied classes receive the same allocation relative to their weights. Equation 2.7 ascertains that the capacity is entirely used [19]. Both rules apply to $i \in \mathcal{L} \cup \mathcal{I}$. In Chapter 5, we describe and implement the algorithm that computes the HMM-fair allocations of the scheduler to the extent that the discrete granularity of a byte allows. We jumpstart this development in Section 5.1 by writing a general HMM module separate from the scheduler. That enables us to calculate fairness metrics and obtain allocation guarantees using a weighted hierarchy and a resource's capacity alone.

$$a_i < r_i \Rightarrow \frac{a_i}{w_i} \geq \frac{a_j}{w_j}, \forall j \in sib(i) \tag{2.6}$$

$$\sum_{j \in \mathcal{L}} a_j = \min(\sum_{j \in \mathcal{L}} r_j, C) \tag{2.7}$$

Figure 2.7 introduces guarantees g alongside weights. Class Y, for instance, got 60 more than its guarantee of 300. Class X and its children used less than they could, so the excess got redistributed. These global guarantees can be derived from the weights and

the capacity with Formula 2.8, explaining how, e.g., classes X_1 and X_2 obtained their guarantees.

$$g_i = C \cdot \prod_{\substack{j \in anc(i) \cup \{i\} \\ j \neq \text{root}}} \frac{w_j}{\sum_{k \in sib(j)} w_k} \tag{2.8}$$

Guarantees can be viewed as weights as long as the property $g_i \geq \sum_{j \in child(i)} g_j$ holds, where $g_{\text{root}} = C$. Throughout this thesis, we assume that to be the case and interchangeably work with $w_i = g_i$, since we are interested in distributing a minimum guaranteed number of bytes per scheduling round. Above that, however, is that the HLS scheduler does the same: closely following their design eases the porting process as described in Section 5.2. In a real-world deployed Fair Stream Scheduler instance, however, ratios at the sibling level would be more intuitive for application developers, abstracting the scheduler's intricacies away and more closely resembling HTTP/2 weights.

Chapter 3

Related Work

This chapter offers a concise literature review of the most promising scheduling approaches identified for integration into our scheduler.

3.1 Multipath QUIC Schedulers

This section provides the necessary context to understand the operation of the SA-ECF scheduler that our implementation utilizes. It is the stream-aware variant of ECF, which, in turn, improves on LowRTT and is the default quiche scheduler. Nonetheless, for the sake of completion, we also present viable alternative approaches, with Chapter 4 explaining why we opted not to use them in our implementation.

A multipath scheduler is a component in the implementation of a multipath transport protocol such as MPQUIC and MPTCP. Scheduling decisions are made per connection and heavily influence their performance [4]. The *packet scheduler* decides which path to use for transmission; in QUIC, packets reliably carry in-order data with `STREAM` frames [1]. The *stream scheduling* component, on the other hand, decides how to produce this output. A scheduler is *stream-aware* if its algorithm gleans information from connection-level stream metadata to make optimal decisions regarding ordering, path choice, and allocation [10].

3.1.1 Earliest Completion First

The path heterogeneity in the example outlined in Figure 2.1 is so detrimental to LowRTT's performance that electing to exclusively send on the terrestrial link would have had all packets delivered in order and acknowledged in about 30 ms, less than the

satellite path's flight time. Given that the problem worsens with increasingly asymmetric paths, the Earliest Completion First (ECF) scheduler can choose to *not* send on a path if waiting for a better one to become available results in an earlier completion time. Hence, slow paths are only used if they do not delay a flow's completion. The completion time is estimated from the path's RTT, the RTT's standard deviation σ, and the path's congestion window. The number of bytes left to complete the transmission and a hysteresis constant β are taken into account, too [10]. In MPTCP, the hysteresis constant influences how willing the scheduler is to switch back to using the slower path after deciding it will wait for the fast one to be available [11]. Changing between these two states too often may not be optimal, but so may insisting on the wrong choice for too long be. Striking the right balance between not regretting a right decision too early while ensuring no sunk costs are incurred can be experimentally determined.

Stream-Aware Earliest Completion First

The SA-ECF scheduler is intended for situations where a connection supports the concurrent transmission of multiple streams. It aims to minimize the estimated completion time of individual streams rather than the connection as a whole [10].

The procedure gauges how many bytes k every stream sends before the currently evaluated stream can complete transmission. It is a separate value from the amount of bytes left to send on a given stream and determines whether to wait or send on a path. This "byte count until completion" k is calculated with the help of the gap length g, standing for how many sending opportunities on average other streams get compared to the stream under evaluation. Per send opportunity, L bytes are sent. The required number l of sending opportunities left for a stream is determined by dividing the remaining request by L. A gap arises between them, during which other streams can send. The gap length, therefore, relies on the normalized weights w obtained from an HTTP/2 dependency tree. Relative to the root node, the normalized edge weights determine the bandwidth allocation of each stream. If a stream s is supposed to receive $w_s = 2/3$ of the bandwidth, $1/3$ remains for all others. The stream gets twice the turns others do, i.e., for each of its sending opportunities, the rest is allocated $\frac{1/3}{2/3} = 1/2$ of that [10]. Formula 3.1 states how g is computed.

$$g = \frac{1 - w_s}{w_s} \tag{3.1}$$

As the scheduler interleaves the sending of s and further streams, $g \cdot L$ bytes are, on average, sent in the gap between s. Figure 3.1 illustrates this example where the stream s in blue has a sending opportunity L of 1000 B and a remaining request of 5000 B. As a

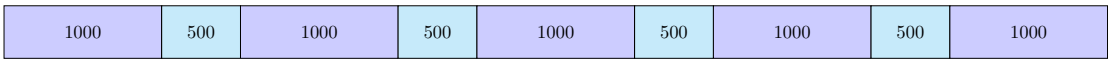

Figure 3.1: Transmission gaps in a Stream-Aware ECF flow

result, it carries this transmission out over $l = 5$ visits. The $l - 1 = 4$ cyan gaps add up to 2000 B in the meantime since it is only granted half of the opportunity. The stream s completes after $k = 7000\,\mathrm{B}$ have been sent over the entire connection. Formula 3.2 provides the general solution for the parameter [10]. In Listing 2, we calculate k using HLS weights instead of HTTP/2 priorities.

$$k = L \cdot (g \cdot (l - 1) + l) \tag{3.2}$$

The transmission time can be approximated with k. Waiting until a congested fast path becomes available to then transfer k packets on entails first waiting for an RTT to let acknowledgments arrive that reopen the path's congestion window `cwnd`. The total estimated completion time is $\mathrm{RTT} + \frac{k}{\texttt{cwnd}} \cdot \mathrm{RTT}$, i.e., the initial waiting time plus however many data bursts are needed times the required timespan. If that takes less than the RTT of another path with enough `cwnd` for k bytes, staying with the blocked path terminates earlier than what can be achieved through switching per the LowRTT strategy. Only if it is greater or equal can the increased aggregate bandwidth provided by a multipath protocol extension decrease the overall completion time [11].

Compared to its default scheduler, experimental results with real networks by Lim et al. [11] found that video streaming bitrate improved by 16% when using MPTCP-ECF in heterogeneous path environments. Full-page web download completion times fell by 26%, and OFO delay by a staggering 71%. In homogeneous scenarios where available path bandwidths and RTTs were roughly equal, ECF's performance matched that of LowRTT. Rabitsch et al. [10] see even further improvements in the order the data is sent due to the stream-aware nature of their scheduler. Their findings suggest that, when compared to plain ECF and LowRTT, SA-ECF is capable of speeding up the stream completion time even in symmetric scenarios.

Shortest Remaining Processing Time

The Shortest Remaining Processing Time (SRPT) scheduler is yet another ECF-based scheduler. It addresses the shortcomings caused by the underlying presence of RR-like approaches in schedulers such as SA-ECF that ultimately depend on a WRR over a HTTP/2 dependency tree. SRPT does not rely on one. Moreover, it achieves a provably optimal completion sequence that is stable: The order in which streams are scheduled

does not change. If desired, it can be modified to accommodate strict priority schemes as in Subsection 2.1.3.

The researchers optimize the Application Data Unit (ADU) completion time such that data on all paths complete simultaneously on the receiver side. The sizes of these *atomic messages* are known by the scheduler in advance, which may not be desirable for continuous data feeds. Additionally, their network model uses fixed values for the path's RTT and the bottleneck capacity rate in bytes per second, which they recognize is not realistic. They find that for any two paths, there is a threshold ADU size at which using the second path becomes useful. As stream sizes increase, also employing larger-latency paths becomes more attractive. The property recursively holds for n paths, yielding $n - 1$ of such thresholds.

The algorithm computes the Shortest Isolated Remaining Completion Time of each message, which is its remaining completion time under ECF if it were not competing with others. Obtained as a function of the path's delay, maximum rate, and ADU size, streams are accordingly sorted in ascending order. Each is assigned an available path with ECF. In a similar fashion to SA-ECF's k from Formula 3.2, the message size is increased to include data scheduled beforehand. Effectively, the completion time of shorter ADUs is minimized by scheduling them first. Longer streams are distributed across slower paths while ensuring simultaneous completion across them.

SRPT-ECF has not yet been implemented within MPQUIC, with trace-based simulations having been done instead. Its online variant is no longer optimal; a concrete implementation must predict when in-flight packets finish on all paths. Benchmarked against SA-ECF, SRPT significantly improved tail stream completion times [8].

3.1.2 Head-of-Line-Blocking Eliminating Scheduler

The Head-of-line Blocking Eliminating Scheduler (HBES) avoids both *inter-* and *intra-stream* HoLB variants. A multipath scheduler is prone to inter-stream blocking if it fails to suitably prioritize streams. Since the images in Figure 2.2 depend on the HTML document to be displayed, a non-optimal transmission order bars the web page rendering process until the markup document arrives. The *inter-* prefix highlights that the blocking arises due to the dependencies *between* streams. If the frames of a stream take separate heterogeneous paths, they are likely to arrive out of order. Stream data must then be rearranged in the receive buffer sorted by their stream offsets, which wastes time and excessively occupies it. An overwhelmed buffer consumes more resources, increases the latency, and may drop packets if full, leading to congestion issues that negatively affect transmission. Since this HoLB delay type is caused *within* a stream,

it is referred to with the *intra-* prefix. HBES consists of two distinct components that independently address these issues for HTTP/2 over QUIC: the Priority-Based Stream Manager (PSM) and the Stream-Aware Arrival-Time-Based Path Selector (SAPS).

Xing et al. conclude that "HBES and SA-ECF bring significant benefits for small streams whose flow completion time is relatively small", with HBES performing "slightly better in asymmetric scenarios." [5]

Priority-Based Stream Manager

The authors improve the WRR scheduler to mitigate inter-stream HoLB. Their central insight is that in window-size-constrained scenarios, low HTTP/2 priorities can still exhaust the sending window despite the presence of higher-priority streams that may only need to send a few packets. Let us assume a WRR scheduling cycle is limited to sending 32 packets. If two streams with priorities of 32 and 128 request 32 and 2 packets, respectively, the higher-priority stream may be blocked. Reiterating Subsection 2.1.1, this is due to the amount of packet-sending opportunities matching the stream's priority without further regarding its position in the scheduling round. Going first, the low-priority stream wholly fulfills its request, blocking its higher-priority sibling.

Their proposed *scattered* WRR emits packets in the ratio of their priorities. The bandwidth sharing proportion of 32:128 is rescaled down to absolute permits of 1:4 in terms of transmission tokens. They allow for the sending of a packet and are consumed as packets are emitted. PSM starts a new scheduling cycle when no HTTP/2 node has any token balance left. At a minimum, streams are assigned at least one token per round. The short rounds help prevent an early exhaustion of the send window and mitigate inter-stream HoLB as a result [5].

As proposed, the PSM is coupled to HTTP/2: usage of the scattered WRR in combination with, e.g., HTTP/3's extensible priorities would ignore the `i` flag. Tied incremental streams, however, could consider PSM's scattered approach.

Stream-Aware Arrival-Time-Based Path Selector

SAPS is intended to mitigate OFO packets even in highly heterogeneous network conditions. A packet is sent on the path for which its estimated arrival time is lowest. The arrival time is the sum of the packet's queueing delay and how long it spends in flight (approximated by halving the RTT, an idea we use in Listing 3). The packet's queueing time is different per path because SAPS requires one send buffer per path, which differs from the shared buffer kept by quiche. The queueing time is the buffer's current occupation plus the packet size to be scheduled, divided by the path's throughput.

SAPS only switches paths when the queueing times become too long, buffering data on low-RTT paths until the queueing time becomes too long, at which point it switches to another path. The path utilization is not maximized, but frames from the same stream will tend to arrive at the receiver in order, and the algorithm can adapt to sudden network condition changes by tracking current queue lengths and path RTT's.

3.2 The Hierarchical Link Sharing Scheduler

The HLS singlepath packet scheduler provably guarantees traffic classes a fair share of a link's bandwidth. Classes can be arranged hierarchically for fine-grained traffic control, with the root stating the available capacity to be distributed to its children. Children can be recursively thought of as the root of a new sub-hierarchy. Packets of differing sizes needing transmission map to childless leaf classes in the base case.

Figure 3.2 illustrates how a link with a capacity of 1000 Mbit/s is subdivided into classes A, B, and C. Each node has a rate guarantee that must be met. A, for example, is guaranteed 300 Mbit/s, but it itself allocates bandwidth to two deeper traffic classes, namely A_1 and A_2. If all leaf classes at the bottom of the hierarchy demand bandwidth above their minimum rates, the sum of the requests is greater than the capacity, limiting them to their guarantees. Likewise, if every leaf requests less than what they are entitled to, their allocated rates match their request.

The difficulty lies in determining a fair bandwidth allocation when the aggregate capacity is exceeded despite some leaves staying under their guarantees. A class' unused capacity can then be fairly redistributed to those that remained unfulfilled. To that end, Luangsomboon et al. developed a computationally low-complexity algorithm [19] that calculates HMM fair allocations. Previous algorithms were prohibitively expensive, partly explaining why the Hierarchical Token Bucket (HTB) and Class-Based Queueing (CBQ) QDiscs saw deployment in Linux. However, these fail to fully isolate rate guarantees from other branches in the hierarchy, which is unfair because classes can craftily change their subtrees to obtain more bandwidth.

The authors deploy a HLS QDisc using a WRR [20]. The scheduler iterates over a set of *backlogged* leaf classes in no specific order, i.e., classes that have packets waiting to be sent. The hierarchy does *not* set out to address dependencies between traffic classes as HTTP/2 trees do. The focus lies on the amount of data a class can send: HLS classes have a balance in bytes instead of sending opportunities. When visited in the RR, the leaves recursively ask for and receive HMM-fair quotas of balance from their parents; the root accrues the balance consumed by leaves when sending packets. A leaf keeps emitting packets while the packet sizes are smaller than or equal to the leaf's balance,

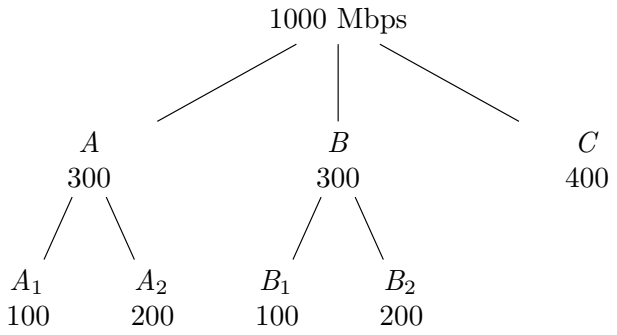

Figure 3.2: A class hierarchy used by the HLS scheduler with rate guarantees [19]

which respectively decreases by that amount. Unused balance is returned to the parent so that siblings of satisfied classes can spend the excess balance first. If they are all idle, the surplus continues traveling upward to the grandparent, ultimately reaching the root.

In Chapter 4, we implement the algorithm for HMM fairness and HLS's methods for QUIC streams instead of packets. They provide transmission quotas given a send quantum and the hierarchy's weights. A further theorem is used to compute an upper byte bound α of how unfair the scheduler's transmission can be in a given time interval.

3.3 Summary

As established in the previous sections, the interplay of the packet- and stream scheduling components can be complex. Table 3.1 outlines elicited requirements toward our scheduler; Chapter 4 provides an in-depth discussion of the necessary software architecture required to fulfill them.

Requirement	**Rationale**
Multipath scheduling	Aggregate the bandwidth of multiple paths
Transparent to the application	Protocol-agnostic, no traffic pattern coupling
Stream-aware scheduling	Reduce Head-of-Line Blocking
Minimum guarantees	Prevent traffic flow starvation
Strategy-proof fairness	Provably prevent manipulation
Hierarchical bandwidth sharing	Control at multiple aggregation levels
React to network condition changes	Paths can be highly heterogeneous

Table 3.1: Elicited requirements for the Multipath Fair Stream Scheduler

Table 3.2 gives an overview of the discussed schedulers. Together with the solution approach, they provide the building blocks of the multipath scheduler built in Chapter 5.

Scheduler	Protocol	Input	Goal	Performance Metric	Fairness	Versatile
ECF	MPTCP	Path's RTT, `cwnd`, σ, bytes left	Maximize throughput	Connection compl. time	No mechanism	✓
SA-ECF	MPQUIC	HTTP/2 tree + ECF inputs	Fast web page load	Stream compl. time, OFO	HTTP/2 WRR	×
SRPT-ECF	MPQUIC	ADU size, path RTT and rate	Optimal scheduling	Stream compl. time	Strict priority	○
PSM	MPQUIC	Dependency tree, rescale factor	No inter-stream HoLB	Stream compl. time, OFO	HTTP/2 WRR	×
SAPS	MPQUIC	Path RTT, throughput, buffer size	No intra-stream HoLB	Receive buffer usage	Handled by PSM	×
HLS	Layer 2	Weighted class hierarchy	Minimum guarantees	Upper fairness bound α	Max-min fairness	✓
This thesis	MPQUIC	SA-ECF + HLS ported to QUIC	Minimum guarantees	Stream compl. time, α	Max-min fairness	✓

Table 3.2: Comparison of deployed and proposed schedulers in the literature

Chapter 4

Design

Designing and implementing a general-purpose, stream-aware fair multipath scheduler for QUIC is a creative systems engineering endeavor that requires novel approaches to be solved. For that purpose, deciding on which of the several QUIC implementations needs to be settled first and foremost. A fork of Cloudflare's quiche [9] is one of the "most promising publicly available MPQUIC implementations" [21] alongside XQUIC and Picoquic. Maintained by Quentin De Coninck, one of the researchers who first designed the extension [22], his work reflects the latest state of MPQUIC's active Internet draft [2]. Written in Rust, it is the open-source library we chose to implement our Multipath Fair Stream Scheduler in, given its IETF compliance and up-to-dateness. This decision affects which works we can draw inspiration from to develop our state-of-the-art scheduler.

Ideally, our scheduler would provide SRPT's optimal scheduling, HLS's provable fairness properties, and minimum rate guarantees. Strict priorities are desirable to address the inherent dependencies found in HTTP traffic and reducing HoLB as SA-ECF and HBES do. These goals, however, conflict, necessitating a decision on which trade-offs to accept.

The fact that HTTP/2 priority trees have been deprecated [14] and only saw mainstream adoption in Firefox [10] is an argument in favor of ensuring that our scheduler can accommodate a wide application range. How protocols develop and are adopted cannot be predicted; tying our scheduler to HTTP/3 could prove erroneous despite it playing a prominent role in the modern IP+UDP+QUIC stack. Even using the same protocol can lead to wildly different traffic patterns, all of which the scheduler should be capable of handling. One way to formalize different traffic patterns is in terms of the transmission's utility over time. For example, a 1 GB bulk file transfer provides

no value until the last byte is reliably received since a corrupted or partially received document is useless. In contrast, an application sending out small Application Programming Interface (API) requests at regular intervals or a video stream sent at 30 frames per second yields a staircase-like utility plot. Testing and evaluating the scheduler in different conditions and scenarios in Chapter 6 therefore requires the development of a flexible traffic generator for the QUIC endpoints to use. The multipath quiche fork ships with simple client and server applications capable of making many simultaneous `GET` requests for files over HTTP/3. They can serve as the basis for a protean traffic generator that uses our scheduler.

ECF is versatile and improves on LowRTT, but a stream-aware variant is needed to avoid HoLB and fully leverage QUIC's multiplexing capabilities. SRPT is a good candidate as it is rather protocol-agnostic. Moreover, it could be integrated with a strict priority scheme if necessary. However, its network model does not yet account for the rapidly changing network conditions we ought to support and explore. Hence, HBES and SA-ECF remain as more promising candidates for real-world deployment.

Rapid deployment is a cornerstone of QUIC's protocol design, explaining why it runs in user- rather than kernel space [23] after learning from how TCP ossified. In a similar vein, the time constraints of this thesis require fast-paced software development practices while ensuring that the code runs properly outside of a research context. The final result should not deviate too much from quiche's existing architecture so that code can readily be understood and contributed back as pull requests. Thankfully, quiche's code quality is high, providing extensive documentation, comments, and over 520 tests that enable a test-driven development approach. Initial experiments with a HBES implementation showed that their requirement of a per-path send buffer alters quiche's inner working more than SA-ECF, despite both fully complying with the QUIC specification.

What we can reuse from HBES, however, is the software engineering principle of separating concerns. Their approach of splitting the scheduler into two separate components, a stream and a path scheduler, is accomplished with the PSM and SAPS algorithms. This good practice already starts to show its value by carving out a simple interface our components need. Instead of receiving the next stream from an HTTP/2 application, another prioritization scheme can feed the available streams into SA-ECF, responsible for deciding which stream goes onto which path next. The process relies on knowing how stream data is interleaved to calculate k, i.e., how many bytes are sent in total for a given stream to complete [10]. Hence, Formula 3.2 will need to be adapted accordingly and by extension Formula 3.1 to determine the gap length g.

Concerning the stream scheduler component, HLS provides a groundbreaking bandwidth-sharing approach that can be adapted to MPQUIC. To the best of our knowledge, this has not been done before. HLS operates with byte-level granularity despite being intended as a packet scheduler, which suits QUIC streams due to their less discrete nature. Its properties go beyond fairness: the authors also prove an upper bound for how long a leaf class has to wait until it is visited by the WRR again [17]. Since HLS does not establish a fixed sending order, SA-ECF can determine the stream ordering. Quiche determines which path a packet takes next to reach its destination address before its construction, so HLS can be used to determine *how much* data to allocate a stream. That amount is HMM fair and sent out in multiple `STREAM` frames. These allocations are strategy-proof and cannot be manipulated, unlike those made by HTB and CBQ [19].

Effectively implemented as an advanced WRR, HLS prevents starvation since a rate guarantee greater than zero will be addressed in a scheduling cycle, no matter how small. Although this excludes a strict-priority approach, it still is an effective means of prioritization: the hierarchy can be set up in such a way that more important classes have the right to more bandwidth. Fairness complements prioritization rather than excluding it by ascertaining traffic flows of equal priority get equitable service, while low-priority flows are not completely service-deprived due to higher-priority flows consuming the available resources entirely [18].

Figure 4.1 gives a high-level overview of the plan outlined in this section. In our design, a QUIC endpoint specifies a hierarchy of traffic classes that reflect the application's requirements. When creating the hierarchy, the application uniquely assigns each stream to a leaf in the hierarchy. Quiche implements an unreliable datagram extension to QUIC [24] that has been formally standardized as a RFC by the IETF QUIC Working Group, but we constrain ourselves to streams. These are depicted as the green, yellow, and green bars entering the QUIC connection from the application. Quiche's connection API is modified to allow the hierarchy to be set from the application layer by passing it as an argument in the call. The hierarchy is stored at the connection level, so a server managing many connections can work with a different hierarchy for each of them. A sensible default can be configured if no hierarchy is specified. We automatically use a flat hierarchy whose streams range from 0–50 with equal guarantees to avoid modifying hundreds of quiche tests with the setter.

The hierarchy is not shared across QUIC deployments, easing interoperability. Opened bidirectional streams can be scheduled differently depending on the connections' vantage point. A quiche-based server may be using a different hierarchy than its quiche peer, and beyond that, the deployment could be talking to XQUIC or Picoquic instances that do not implement this scheduler.

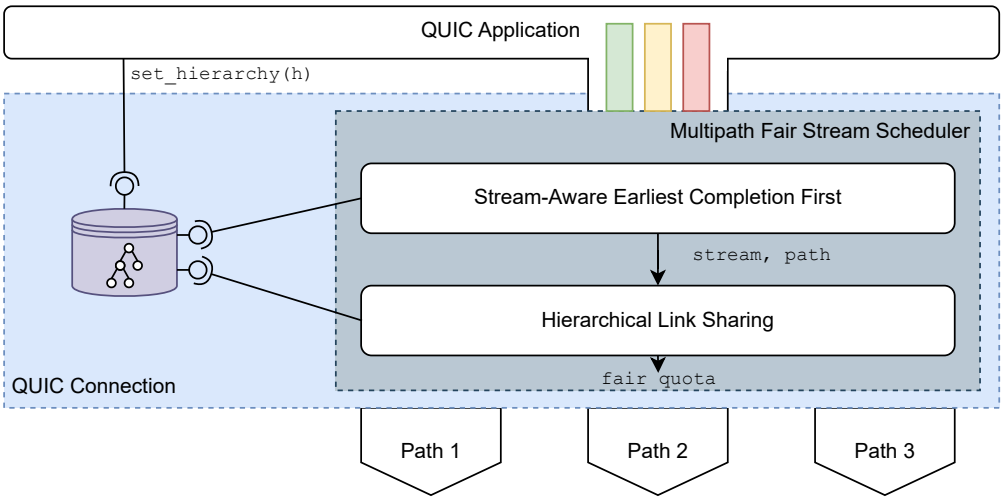

Figure 4.1: High-level architecture of the Multipath Fair Stream Scheduler [5]

With the information stored in the hierarchy, the SA-ECF component decides which stream to schedule and on which path next. In singlepath scenarios, the entire component is irrelevant since there is only one path by definition and a random permutation of the stream ordering is valid for HLS as long as they have pending data to send. When the SA-ECF component is skipped, we retain the ordering given by quiche's stream scheduler for backward compatibility.

The HLS component calculates the stream's fair quota in bytes for that scheduling cycle, whose duration is unspecified as it depends on the guarantees. Designating that component as a *link sharer* rather than a *bandwidth sharing* algorithm is a misnomer, but we keep it to ease understanding and reinforce the source of the principle behind it. The same applies to our modified SA-ECF, albeit to a lesser extent. With the mechanisms for the path choice, stream selection, and byte allocation determined, the implementation of the Multipath Stream Scheduler can begin.

Chapter 5

Implementation

To guarantee streams are allocated a fair portion of the bandwidth, we solve the more straightforward singlepath case before moving on to the multipath context. As outlined in our solution approach, implementing the HLS and SA-ECF components separately is possible because the path is trivially given in a singlepath transmission, and HLS's stream ordering is arbitrary. As a result, the code in Sections 5.1 and 5.2 does not rely on SA-ECF's output.

5.1 The Hierarchical Max-Min Fairness Module

The HMM module was implemented as an experimental sandbox to implement the algorithms of the HLS paper. It helped to understand the material and was used to generate the values of the hierarchical figures in this thesis. This initial coding phase established the concrete data structures to be used in the scheduler's implementation.

A hierarchy, for example, is instantiated as a Rust struct storing the ID of the root node, the capacity, which ID to use next, and the set $\mathcal{N}$. IDs are assigned incrementally from zero. $\mathcal{N}$ is a hash table using the class ID as a key mapping to another struct of HMM classes. Other than the ID, these contain fields for the allocation, resource request, fair share, weight, guarantee, a vector containing the IDs of its children, and optionally the parent: for the root, the parent is `None`. The hierarchy from Figure 2.7 can be specified as per the instructions in Listing 1.

```rust
// Create a new hierarchy, passing the capacity as an argument.
let mut tree = HMMHierarchy::new(1000.0);

// Insert the nodes, passing the request, weight, and parent ID.
let root_id = tree.insert(0.0, 1, None);

let x = tree.insert(0.0, 5, Some(root_id));
tree.insert(150.0, 3, Some(x)); // x1
tree.insert(250.0, 4, Some(x)); // x2

tree.insert(600.0, 3, Some(root_id)); // y

let z = tree.insert(0.0, 2, Some(root_id));
tree.insert(800.0, 1, Some(z)); // z1
```

Listing 1: Specification of a hierarchy

We implement the HMM algorithm using the pseudocode presented in the original HLS papers [19] [17]. It heavily relies on the operations from Table 2.1. We validate the implementation with 15 tests that assert whether properties such as Rule 2.6 and 2.7 hold. Another is that even though only nodes in $\mathcal{L}$ make requests, the resulting allocation follows Formula 2.5. By calling `hierarchical_max_min_fair(&mut tree)`, the HMM-fair result is returned and displayed in Listing 2 with a debug `impl`. Figure 2.7 provides a graphical representation.

```
0: C: 1000.0, r: 1800.0, a: 1000.0, w: 1, g: 1000.0, f: 120.0
|- 1: r: 400.0, a: 400.0, w: 5, g: 500.0, f: inf
|  |- 2: r: 150.0, a: 150.0, w: 3, g: 214.29
|  |- 3: r: 250.0, a: 250.0, w: 4, g: 285.71
|
|- 4: r: 600.0, a: 360.0, w: 3, g: 300
|- 5: r: 800.0, a: 240.0, w: 2, g: 200.0, f: 240.0
   |- 6: r: 800.0, a: 240.0, w: 1, g: 200
```

Listing 2: Sample output of the HMM algorithm

Taken literally in a scheduling context, Listing 2 could be problematic due to the presence of floating point numbers in the guarantees for the classes 2 and 3, i.e., X_1 and X_2. Packets, not fractions of a byte, can be scheduled with the HLS scheduler. This

inherent numerical limitation affects how fair a practical scheduler can be in contrast to the principle of HMM fairness implemented by this module.

In an arbitrary period, the weighted difference between how many bytes two classes transmit should be as close to zero as possible. Let $D_i(t_1, t_2)$ be the number of bytes a class i transmits in the interval $[t_1, t_2]$. The left side of the Inequality 5.1 then provides the deviation from an ideal HMM scheduler [19].

$$\left| \frac{D_i(t_1, t_2)}{w_i} - \frac{D_j(t_1, t_2)}{w_j} \right| \leq \alpha \tag{5.1}$$

The parameter α on the right side is an upper bound for how badly the HLS scheduler can perform given a hierarchy and maximum packet size $L_{\max}$. The HLS paper [19] provides an equation to compute it, with the proof being provided in [17]. Our HMM module outputs Listing 3 in the console for the hierarchy in 3.2 with a maximum packet size of 1500 B, matching the expected result according to the papers. It also indicates one pair of siblings with this largest deviation, corresponding to A_1 and A_2 in this case.

```
1 alpha(HLS) is 103.5 for nodes i=4, j=5 and leaf max data of 1500
```

Listing 3: Console output displaying the fairness bound α of a hierarchy

With the solution approach and the theoretical framework for fairness in place, we can being implementing our scheduler.

5.2 Porting the Hierarchical Link Sharing Scheduler

To apply HLS in MPQUIC, we alter the algorithm used in its QDisc implementation [20] to encompass QUIC's stream multiplexing capabilities. The QDisc differs from the optimal HMM allocation, sacrificing, for example, floating points operations in the Linux kernel for performance and not implementing the surplus rounds. We implement them as described in the HLS paper, practically validating the theorized approach regarding the distribution of excess capacity.

5.2.1 Start of a Scheduling Round

Our scheduler takes hold once quiche has included the necessary control frames in a packet and is ready to decide how to use the remaining packet space for a single `STREAM` frame. A scheduling round consists of a *main* and one or more *surplus* rounds. The HLS scheduler instance visits streams in the order specified by a queue of pending leaf

classes. The head of the queue indicates the stream whose payload will be included next since we give leaf classes an attribute for the stream ID. This round-robin progresses to the next stream by dequeuing the element at the front once the stream has either fulfilled its request, or there is no balance left to send.

When there are no pending leaves to visit, the scheduler is configured to start a new round either again or when it is initialized. To that end, we get the set $\mathcal{L}_{\text{ac}}$ of active backlogged classes, i.e., the streams that quiche deems flushable due to having available data to send. The scheduler iterates over this set of pending leaves in main or surplus rounds. The ancestors of an active leaf class are also deemed active, forming the set of active internal nodes $\mathcal{I}_{\text{ac}}$ (which excludes the root node). The fair shares f of nodes in $\mathcal{I}_{\text{ac}} \cup \{\text{root}\}$ are reset, as they are recomputed every round.

The HMM allocation of a node i is given as a balance B_i. Additionally, active internal nodes and the root keep track of a *residual* R_i that collects unused balance during a round. At the start of a round, these residual transmission permits become proper balance with the update given by Formula 5.2 [17].

$$B_{\text{root}} \mathrel{+}= R_{\text{root}}, \;\; R_{\text{root}} = 0 \tag{5.2}$$

5.2.2 Selection of the Round-Robin's Quantum

When the scheduler is initialized, balances and residuals are set to 0. Only the root is assigned a balance B_{root} of Q^* bytes. This quantum is the maximum number of bytes the scheduler can transmit in one of the round types [17]. A large quantum causes rounds to take too long, making the scheduler react slowly to inactive streams becoming backlogged. On the other hand, too short of a quantum sends plenty of overhead for a few payload bytes.

This problem is different to the one faced by the HLS QDisc, where the risk is "looping indefinitely [...] without any transmission" [19], since the balance passed down from the root could be smaller than the size of the packet wanting to be transmitted. To avoid this issue, the authors select Q^* such that at least one packet is transmitted per main round. That entails dynamically adjusting Q^* at the round's start depending on which leaves became or stopped being active. The quantum is accordingly reduced or increased by a maximum packet size L_{max} set to 1500 B.

We have a finer granularity thanks to using streams, but an analog to L_{max} is still required to derive Q^* and the fairness bound. Instead of dynamically adjusting Q^* with a maximum packet size, streams add or subtract their minimum guarantees.

$$Q^* = \sum_{i \in \mathcal{L} \cup \mathcal{I}} g_i + \sum_{i \in \mathcal{L}_{\text{ac}}} g_i \tag{5.3}$$

The first sum is constant in Formula 5.3 as the hierarchy is static once set by the application. It ensures the root gets as much balance as needed to satisfy any leaf or internal class. The second sum is dynamic as it checks which streams have outstanding data to send. When a stream i becomes active at the start of a round, we add g_i to the root's residual. If it became idle in the previous round, we subtract g_i.

The second term is at most equal to g_{root}. Per Section 5.1, the root's guarantee equals the capacity C in the HMM context, but our scheduler's quantum is dynamic and may be larger than that. Instead of completely disregarding the capacity parameter, we use it to limit the maximum amount of data a stream can emit in a round, regardless of how much balance it has left. The root's guarantee is hence used as the maximum stream length $L_{\max}$. When calculating fairness bounds, our HMM module produces a warning if not.

In the HLS paper, the Invariant 5.4 is provided. The property formally validates our implementation, as our scheduler intentionally panics if the quantum is not the sum of the available and unused balances.

$$\sum_{i \in \mathcal{N}} B_i + \sum_{i \in \mathcal{I} \cup \{\text{root}\}} R_i \equiv Q^* \tag{5.4}$$

5.2.3 Balance Allocation

With the round initialized, a class at the head of the pending leaves queue is being *visited* by our scheduler. A visit continues over multiple iterations until the HMM balance allocation is emitted or the stream's request is fulfilled. In that process, it emits an unspecified number of packets that carry an unspecified amount of `STREAM` data. The first of such visits within a round is called a *tick*. When a leaf ticks, the scheduler computes the fair quota of the parent, with which the total quota for the leaf in terms of balance is obtained. The update is given by Formula 5.5.

$$B_i \mathrel{+}= g_i f_{p(i)},\ B_{p(i)} \mathrel{-}= g_i f_{p(i)} \tag{5.5}$$

The parent $p(i)$ is in $\mathcal{I}_{\text{ac}} \cup \{\text{root}\}$. Their fair quota, given by Formula 5.6, is only calculated once per round.

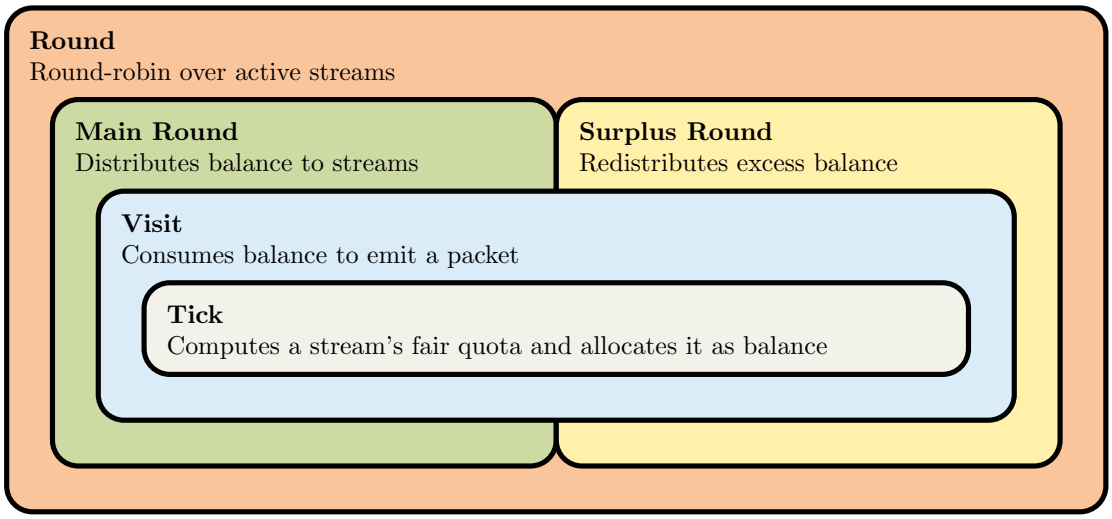

Figure 5.1: Scheduling round terminology

$$f_i = \left\lfloor \frac{B_i}{g_i^{\text{ac}}} \right\rfloor, \; g_i^{\text{ac}} = \sum_{k \in child(i) \cap (\mathcal{L}_{\text{ac}} \cup \mathcal{I}_{\text{ac}})} g_k \tag{5.6}$$

Their balance is updated with Formula 5.7. Note that the recursive process starts from the leaf classes, which request balance based on their parent's fair quota. The request escalates up the hierarchy until the root is reached or some internal node has already computed the fair quota. Balance is then distributed from the top down, completing the tick.

$$B_i \mathrel{+}= (R_i + g_i f_{p(i)}), \; B_{p(i)} \mathrel{-}= g_i f_{p(i)}, \; R_i = 0 \tag{5.7}$$

5.2.4 Sending a Packet

Figure 5.1 gives an overview of the terminology used to describe the states our scheduler can be in. During a visit, the scheduler checks a stream's available balance and how much data it requests. L is the space quiche determined to be left in the packet for `STREAM` data. With e_i, we keep track of how many bytes the stream has emitted to determine its remaining maximum request size. It is reset at the start of a round.

$$a_i^{\text{visit}} = \min\{B_i, r_i, L, g_{\text{root}} - e_i\} \tag{5.8}$$

Quiche then writes a_i^{visit} bytes into the stream payload, and we increase e_i by that amount. The leaf class consumes this balance and returns it to the root.

$$B_i \mathrel{-}= a_i^{\text{visit}},\ B_{root} \mathrel{+}= a_i^{\text{visit}} \tag{5.9}$$

5.2.5 Returning Unused Balance

Should a leaf satisfy its request during a visit, it becomes idle and returns the remaining balance to its parent. Only the root cannot return the balance upstream. The idle class is removed from the set of active classes; its unused balance is added to the residual of its parent.

$$R_{p(i)} \mathrel{+}= B_i,\ B_i = 0 \tag{5.10}$$

An internal class becomes idle when all its children are. Formula 5.10 is recursively applied until a class needing the balance or the root is encountered. After the update, the class has no balance left, and the round-robin is advanced by popping the queue's front off.

5.2.6 Start of a Surplus Round

A new main round starts if the previously distributed quotas have entirely been used for transmissions. The only circumstance under which this does not happen is if a stream became idle in the last iteration. In that case, the condition $B_i + R_i \geq g_i^{ac}$ is met for at least one internal class, and a surplus round begins.

The only difference between a surplus and a main round is that the root's fair quota f_{root} is set to zero [19] instead of `None`, which prevents it from distributing new quotas. The same approach is used should the root's balance ever be negative after the changes from Subsection 5.2.2 regarding the root's residual and subsequently Formula 5.2. As a result, only excess capacity is distributed to newly backlogged or remaining streams. Once the surplus round terminates, more follow until the condition stops holding.

5.3 Porting the Stream-Aware ECF Scheduler

Although we do not know how much each stream emits per visit, we know that it is allocated at least its guarantee per round. We port the SA-ECF path selection procedure described in Subsubsection 3.1.1 by estimating the completion time of the streams in terms of scheduling rounds rather than individual sending opportunities.

Algorithm 1 provides the ported SA-ECF algorithm that relies on the ancillary method stated in Function 2. The implementation differs from the original [10] in that HLS guarantees are used instead of HTTP/2 weights. We schedule a specific stream by swapping the leaf at index i with the first element of the pending leaves array, changing the round-robin order used by the Fair Stream Scheduler. In the paper introducing the ECF algorithm for MPTCP, the researchers presented their results with $\beta = 0.25$, but find that other values "yield similar results" [11]. We use $\beta = 1$ to avoid floating-point operations. A higher hysteresis constant makes it more likely that a waiting stream will not switch back to the slower path.

The fastest and the fastest available paths are chosen with the LowRTT strategy. Already present in quiche, it aligns with our requirement of straightforward integration in existing QUIC implementations. If the congestion window of the fastest path is open, meaning that the path is available, it is selected for transmission without further considering the stream's ordering.

We recall that if the fastest path is blocked, the LowRTT strategy moves onto the second fastest available path. In contrast, SA-ECF may wait for it to reopen. The pressing question is which stream to send on the fallback path, if any. Each stream decides whether they accept being sent on the slower path or want to wait for the faster path to become available again [10]. With the value k, they estimate the total number of bytes the connection transmits until their request is fulfilled, which may occur over multiple scheduling rounds. Together with the path's congestion window, the total wait and transmission time n for k bytes in terms of the RTT is determined [11].

If waiting for the fastest path and performing the entire transmission over multiple bursts takes longer than sending everything at once on the fallback path, the stream does not wait and is immediately scheduled. Otherwise, a stream is set to wait if the algorithm validates that transferring on the second path does not complete earlier than on the first, which will need at least 2 RTTs: once for it to reopen due to acknowledgments arriving and once for the actual transmission [11].

Since the congestion windows and RTT values vary, the scheduler compensates for this variability using the standard deviations σ of the RTT as a margin [11]. Quiche's connection API provides access to a path map tracking these per-path statistics, which we use to instantiate the pseudocode. We changed quiche's logging mechanism to state which path ID was used when sending a packet.

Algorithm 1 Ported Stream-Aware Earliest Completion First Algorithm [10]

Input: paths, array of leaves $\mathcal{L}_{\text{ac}}$ to visit
Output: the path and stream to use for the next packet
Find the path p_f with the lowest RTT
Find the path p_s with the lowest RTT that is available
if p_f available **then**
    return $p_f, \mathcal{L}_{\text{ac}}$ ▷ Pending leaves untouched
else
    $\beta = 1$ ▷ 0.25 in ECF
    $\delta = \max\{\sigma_f, \sigma_s\}$
    for leaf, i **in** $\mathcal{L}_{\text{ac}}$ **do**
        k = `bytesUntilCompletion`(r_i, g_i)
        n = $1 + \frac{k}{\texttt{cwnd}_f}$ ▷ Wait and transmission time
        if $n \cdot \text{RTT}_f < (1 + \text{leaf.waiting} \cdot \beta) \cdot (\text{RTT}_s + \delta)$ **then**
            if $\frac{k}{\texttt{cwnd}_s} \cdot \text{RTT}_s \geq 2 \cdot \text{RTT}_f + \delta$ **then**
                leaf.waiting = 1
                continue
            else
                return $p_s, \mathcal{L}_{\text{ac}}$.swap$(0, i)$
            end if
        else
            leaf.waiting = 0
            return $p_s, \mathcal{L}_{\text{ac}}$.swap$(0, i)$
        end if
    end for
    return no transmission
end if

Method 2 approximates k. If a stream's request is smaller than its per-round guarantee, it will become idle in the current scheduling round. In that case, it sends its complete request in bytes, assuming it remains at the head of the pending leaves queue. If fulfilling the request requires multiple rounds, we calculate how many bytes other streams are guaranteed in the meantime. We purposely sum over $\mathcal{L}$ to consider all guarantees, even those that are not active, as the guarantees of inactive streams are redistributed.

Algorithm 2 Ported `bytesUntilCompletion` method [10]

function bytesUntilCompletion(r_i, g_i)
  if $g_i \geq r_i$ **then**
    return r_i ▷ Stream i completes in the current round
  end if
  left = $\frac{r_i}{g_i}$ ▷ Remaining rounds
  gap = $\sum_{j \in \mathcal{L}, i \neq j} g_j$ ▷ Guarantees of all other streams
  return (left - 1) · gap + r_i
end function

Listing 3 removes the complexity introduced by β, and fewer opportunities are given to select the slower path. Although MPTCP returns data acknowledgements on the receival path, any can be used in MPQUIC [4] [21]. SA-ECF does not account for this difference. We assume the peer will return acknowledgements on the fastest path in an attempt to better approximate a path's re-opening time.

Algorithm 3 Adapted Stream-Aware Earliest Completion First Algorithm [10]

Input: paths, array of leaves $\mathcal{L}_{\text{ac}}$ to visit
Output: the path and stream to use for the next packet
Find the path p_f with the lowest RTT
Find the path p_s with the lowest RTT that is available
if p_f available **then**
  return $p_f, \mathcal{L}_{\text{ac}}$
end if
$\Delta_f = (\text{RTT}_f + \sigma_f)/2$ ▷ One-way delay with symmetric paths
$\Delta_s = (\text{RTT}_s + \sigma_s)/2$
$\Delta_{ack} = \min\{\Delta_f, \Delta_s\}$ ▷ One-way `MP-ACK` delay
for leaf, i **in** $\mathcal{L}_{\text{ac}}$ **do**
  k = `bytesUntilCompletion`(r_i, g_i)
  n = $1 + \frac{k}{\texttt{cwnd}_f}$ ▷ Number of bursts
  m = $\frac{k}{\texttt{cwnd}_s}$
  if $n \cdot (\Delta_f + \Delta_{ack}) < m \cdot (\Delta_s + \Delta_{ack})$ **then**
    continue ▷ Wait for faster path
  end if
  return $p_s, \mathcal{L}_{\text{ac}}$.swap$(0, i)$
end for
return no transmission

Chapter 6

Evaluation

In this chapter, we outline our experimental setup to assess the performance of the Multipath Fair Stream Scheduler. We conduct the experiments on the Baltikum testbed of the Chair of Network Architectures at Technical University of Munich (TUM) because it enables network experiments to be specified and reproducibly run [25]. Reproducibility is crucial because HLS's scheduling decisions rely on classes having backlogged data at the start of a round. However, generating traffic at a rate that "ensures each active leaf class is permanently backlogged" [17] is highly dependent on the hardware and environment used. For example, the scheduler's initial development was locally done on a personal computer and laptop, where the client and servers communicated over the loopback interface. It rapidly became apparent, however, that this approach was misguided, as it led to differing testbed results. We, therefore, need to establish a controlled environment regarding the hardware, traffic patterns, network topologies, and hierarchies used in our experiments.

The testbed consists of several machines connected in a particular pre-existing topology. We were initially interested in reserving a pair where one node can act as a client and the other as a server. We wanted topologies linked with at least two cables to simulate network routes over paths that are physically, not logically, distinct.

Given these constraints, the Riga-Vilnius nodes were used in early tests due to their low demand arising from hardware limitations. These tests, however, ran into a few problems. It became apparent that having a QUIC client and server run on separate machines led to incorrect timestamp measurements as their clocks were not synchronized. For example, with microsecond precision, some delays were reported as negative, which led us to use a design in which a client on Riga connects to a server on the same

machine. The host, aware of this and intending to use the quickest route, stopped involving the intermediary node in the transmission. We separated the server and client with network namespaces to ensure they cannot communicate directly internally. With IP namespaces, the kernel is artificially forced to send packets over the wire. Once packets reach Vilnius, it acts as a network bridge on Open Systems Interconnection (OSI) layer 2 by simply forwarding the packet back to Riga.

While this fixed the timestamp issue, having both applications run on the somewhat outdated hardware led to low throughput. Recognizing this performance bottleneck, we upgraded to the Klaipeda-Narva pair. The technical details in Table 6.1 provide an overview of the hardware used throughout this chapter. Sections 6.1 and 6.2 delve into the single- and multipath topologies in more detail from the IP layer perspective.

	Klaipeda	**Narva**
CPU	Intel Xeon E3-1230 @ 3.20 GHz	Intel Xeon E3-1230 v2 @ 3.30 GHz
Cores	4	
Memory	16 GB	
Link	10GBase-CX4	
Interface	Intel 82599ES 10-Gigabit SFI/SFP+	

Table 6.1: Hardware specifications for the Klaipeda and Narva testbed nodes

The experiments performed in this chapter rely on the hierarchy specified in Figure 6.1. This instantiation was chosen for its parsimony in illustrating the critical properties of the scheduler, as outlined in the experiments of the original HLS paper. By having class C be a leaf while A and B are internal classes, the hierarchy enables us to validate whether HMM fairness is maintained once C stops. Furthermore, although A and B are equally weighted, their children's guarantees differ significantly. As we will see, that enables us to observe that our scheduler provides class isolation, unlike HTB and CBQ [19].

6.1 Singlepath Experiments

As illustrated in Figure 6.1, IP nodes participate within the `192.168.99.0/24` subnet in our singlepath network. The host is assigned the address `192.168.99.1` on port `4433`, while the client binds to `192.168.99.2` on any available UDP port, such as `25565` (indicated with a zero). We leave the path as-is, i.e., we do not artificially limit the bandwidth or specify a specific loss rate, and the unchanged RTT is approximately 0.18 ms. Given that a 10 Gbit/s link is available and that capacity will not be exceeded, we can perform experiments in two scenarios. The former has the Central Processing

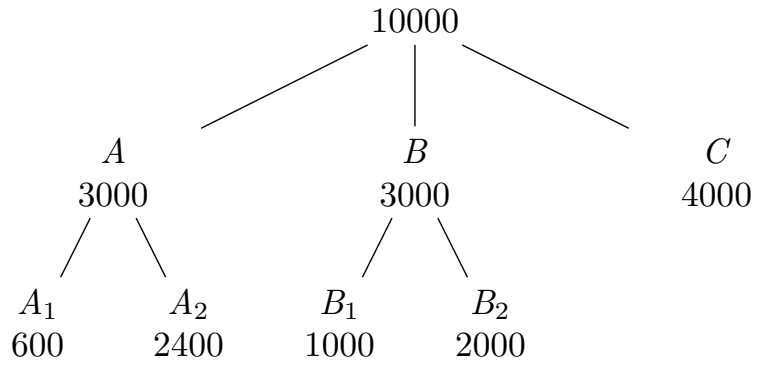

Figure 6.1: Hierarchy used across our experiments

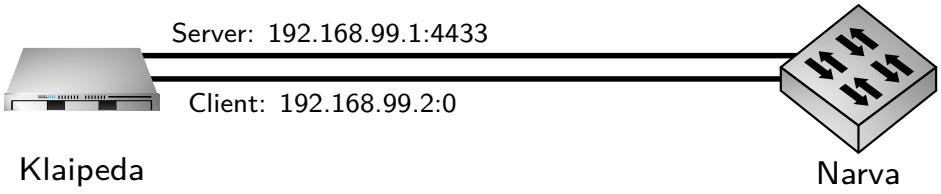

Figure 6.2: Singlepath testbed topology

Unit (CPU) as the limiting factor for the achievable bandwidth; the latter emits at a lower rate than the CPU cores allow. To that end, one of the installed dependencies in Section 6.3 is a performance monitoring tool that logs the resource usage as our experiments run.

6.1.1 Bulk HTTP/3 Transmission

In this experiment, we run quiche's HTTP/3 client and server from the MPQUIC fork. The traffic pattern consists of concurrent bulk transmissions of five 12.5 MB files hosted by the server. Their names match the class hierarchy they're transmitted on, e.g., class A_1 delivers `A1.txt`, whose contents are randomly generated. The client opens bidirectional streams to request the files simultaneously, which the server uses for its response. To ensure prioritization is only handled by the Fair Stream Scheduler, we explicitly set the extensible HTTP priorities to the same value across all streams. Specifically, they are given the highest urgency of 0, and the incremental flag is false. In the outlined scenario, the relevant Key Performance Indicators (KPIs) are the per-stream throughput over time, fairness, and the end-to-end completion time from the instant the client made a request to the moment it was fully received.

The transmission of 500 Mbit worth of data occurs at a rate of approximately 66.75 Mbit/s. If treated as a whole, each stream obtained a bandwidth of 13.35 Mbit/s in that 7.5-

second timespan. Figure 6.3 shows that achieved throughput per class varies significantly throughout the transmission. The plots use bins of 200 ms, following those presented in the HLS paper [19]. A finer granularity of 5 ms is used to analyze the scheduler's behavior in specific areas of interest. Figure 6.4 zooms into the first 0.2 seconds of the transmission to highlight the initialization phase, where the server processes requests as they arrive. Unlike the previous graph, it stacks each class' throughput, with their sum corresponding to the CPU-bound capacity. Figure 6.5 the same visualization technique over the entire transmission.

Class A_1 is the first to arrive and, while alone, uses all available bandwidth despite being the lowest-weighted class. Once its sibling A_2 is backlogged, the scheduler correctly throttles A_1 to a fourth of A_2's bandwidth. We see similar behavior in class B, but when B_2 becomes active, it only consumes the resources its sibling used excessively. This illustrates that the children of A and B are isolated from each other. C affects its siblings once it starts transmission, seen in its nephews adjusting to their minimum guarantees.

Zooming back to Figure 6.3, with all classes backlogged, the minimum guarantees are sustained until C completes its transmission. In the meantime, global weights are satisfied: Class B_2, for instance, receives twice as much bandwidth as B_1 but half as much as class C. When it becomes idle around the 4-second mark, C's unused capacity is re-distributed to the remaining active classes in the hierarchy. Their throughput increases fairly: the previous ratios are kept, e.g., A_2 is still allocated four times as many resources as A_1. It is only when A_2's request is fulfilled that A_1 uses the excess capacity, which would have previously been unfair. Although the node is weighted with 600, it is the sole active node of a 3000-weighted parent, and the child can exceed its guarantee by consuming the entire balance of the parent. Likewise, class B_1 matches this weight of 3000 once B_2 is idle. In the last second of the transmission, class A_1 finishes, briefly letting B_1 receive all available balance until it completes, too.

We use our HMM module to assess the scheduler's fairness formally. It reveals that the hierarchy of Figure 6.1 has an α of 102 B. As long as the weighted transmission difference of two active sibling pairs stays under that upper bound, we can claim the transmission to have been HMM-fair. Using Formula 5.1, we plot the unfairness over time of this experiment's transmission in Figure 6.6 using intervals of 50 ms. We chose A_1 and A_2 because this sibling pair has the alpha with the largest allowed deviation; other pairs have alphas, too, but the hierarchy's alpha is their maximum. For example, classes A and C can only deviate up to 27.5 B and claim to be fair. As the alpha metric is intended for packet schedulers, it provides plenty of leeway in our byte-granular stream scheduler. Except for the intervals during which A_2 either did not arrive or is idle, the

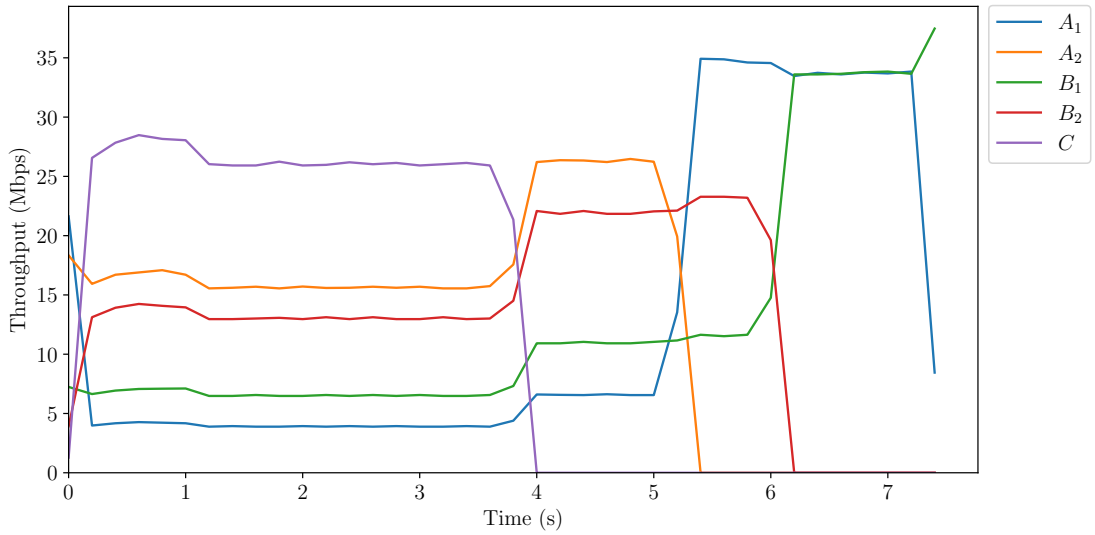

Figure 6.3: Singlepath fair stream scheduling with quiche's HTTP/3 applications

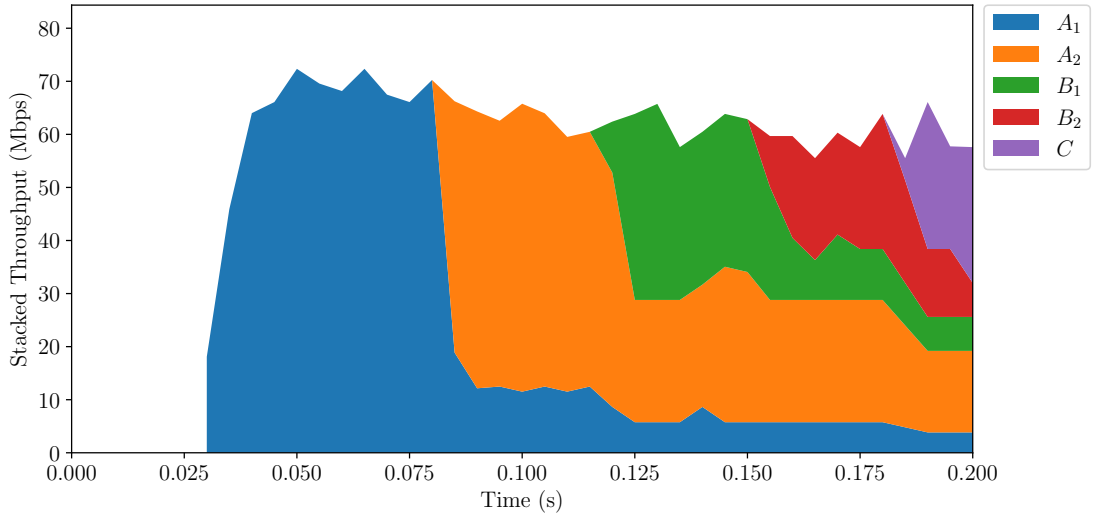

Figure 6.4: Effect of HTTP/3 `GET` requests on the server's stacked throughput

transmission is almost perfectly fair, even when class C completes and re-distributes its capacity. Thanks to class isolation, A_2 consuming A_1's excess balance does not impact the fairness of class A regarding B or C.

6.1.2 Changing Network Conditions

The virtual setup used in the SA-ECF paper uses Netem to limit traffic over a WLAN node to 50 Mbit/s with an RTT of 10 ms [10]. In this experiment, we constrain the run from Subsection 6.1.1 on a real testbed to these path characteristics approximately

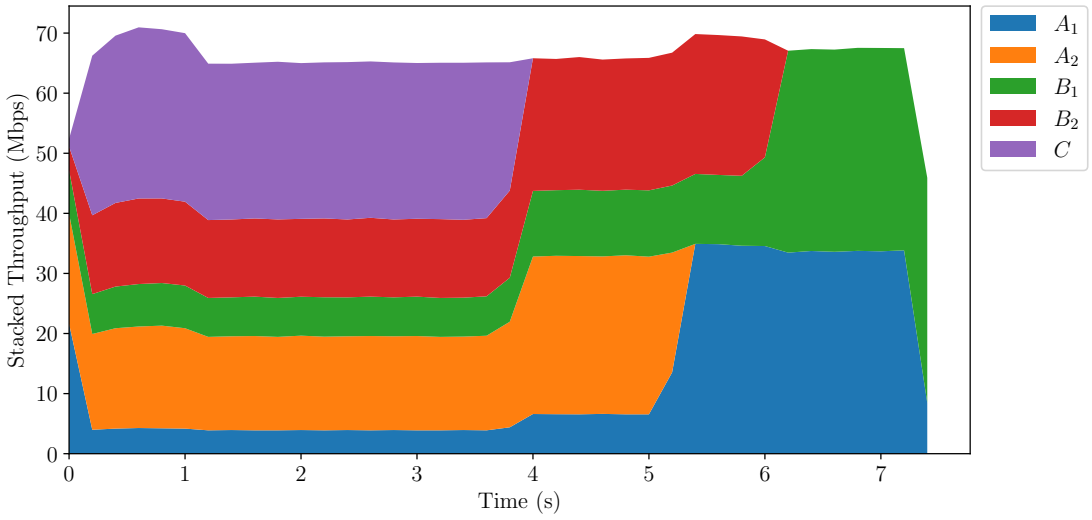

Figure 6.5: Stacked throughput of an HMM-fair transmission

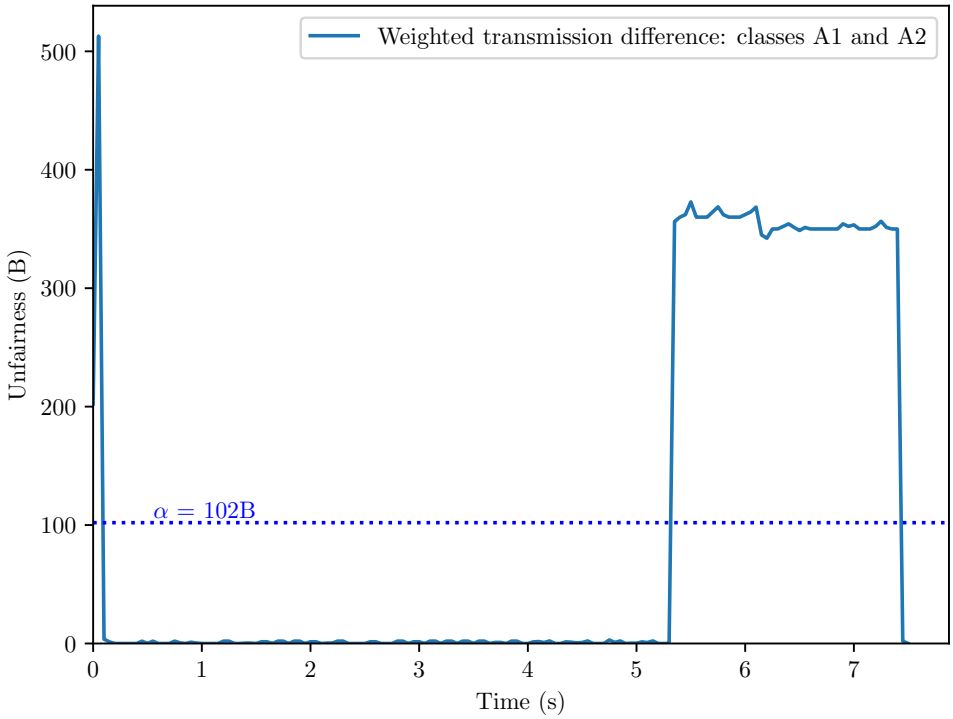

Figure 6.6: Graphical validation of HMM fairness with the fairness bound α

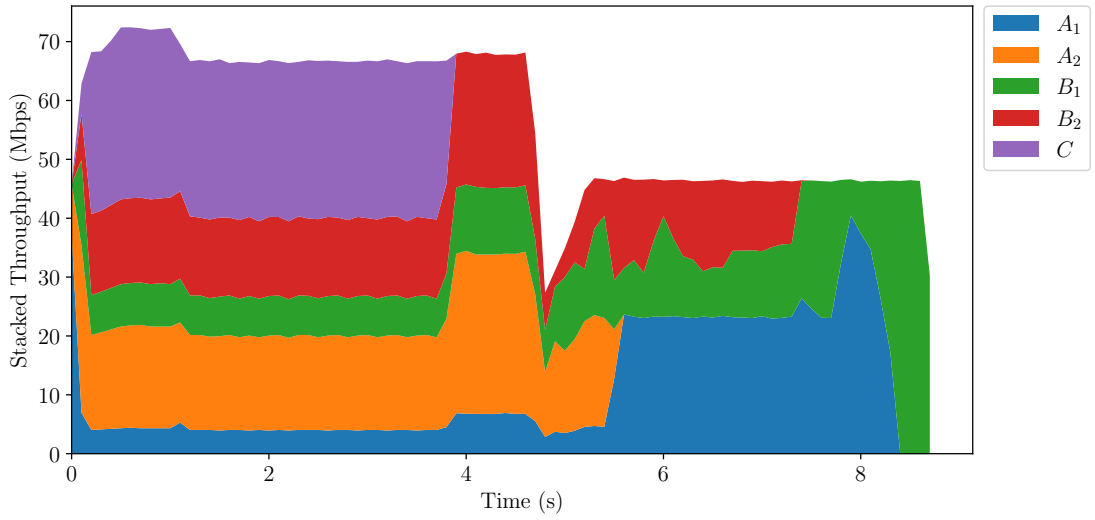

Figure 6.7: Effect of Netem on a HTTP/3 transmission

5 seconds into the HTTP/3 transmission. The constraints are achieved by limiting Klaipeda's outgoing interfaces to 5 ms and 50 Mbit/s, such that bidirectional traffic is equally affected on the way to and from its peer.

Figure 6.7 shows the transmissions' throughput as a stackplot with bins of 100 ms. Overall, the additional bandwidth that classes A_1-B_2 obtain once C terminates is lost with the imposed constraint: their throughput is squished to a rate resembling what they obtained while all streams are backlogged. Furthermore, classes A_1 and B_1 consume their siblings' bandwidth when terminating.

Admittedly, however, the transmission should be smoother. After imposing the Netem bandwidth bottleneck, class B_1 spikes twice. In the first time, it comes close to surpassing the fairness bound α, which it does the second time. Class A_1 has two similar instances: when B_2 finishes, it briefly seems to receive more balance, which should not occur as their parent classes are different. At that point, A_1 and B_1 should split the available bandwidth equally, but A_1 eats into B_1's bandwidth instead.

The problem may lie with the scheduler's quantum staying the same despite the new bandwidth constraint. Although the HLS scheduler is intended to be used alongside traffic shapers, its quantum size arising from absolute weights that match an equal amount of bytes may be responsible for these problems. Using a quantum based on the congestion window size and RTT warrants further investigation. In comparison, tests with Netem specifying large RTT values only, such as 100 ms, caused the startup phase of the scheduler to be slow but with significantly fewer issues regarding HMM fairness.

6.1.3 Protocol-Agnostic Transmission

In this experiment, we use a self-written traffic generator, stepping away from the previous reliance on the HTTP/3 client and server provided by the MPQUIC implementation. We aim to demonstrate that the desirable HMM properties of our scheduler are consistently present, regardless of the application-layer protocol used.

The five leaf classes in this experiment's traffic pattern continuously request 1200 B of stream data. The payload comprises 75 equal 16 B timestamps of when the application gave quiche the data to handle. When received by the client, the total end-to-end latency is logged, i.e., the sum of all delays, such as in-flight or buffering times.

Our setup includes send- and receive buffers that can each accommodate a single request per stream. New data is only generated if a stream backlogs less than this buffer size, ensuring they are permanently backlogged without overburdening the CPU. We employ the `sar` and `pidstat` tools to meticulously monitor system-wide CPU usage, as well as the usage by client and server applications. During a 30-second run, we observe that they are assigned separate CPU cores, with the server utilizing a maximum of 40% of its core and the client 61%.

As expected and shown in Table 6.2, the average throughput is significantly smaller than in the previous experiment. Nonetheless, the hierarchy's absolute weights ensure that the allocated bandwidth closely resembles the desired theoretical result. Figure 6.8 displays an empirical Cumulative Distribution Function (CDF) plot of the end-to-end latency, which confirms that the weights also provide a means of delay prioritization. Higher weights correspond to lower latencies, with a strict separation between classes. Unfortunately, this difference is not proportional, as, e.g., class C has double the weight of B_2 but not half the delay.

				Delay (ms)	
Class	**Throughput** (Mbit/s)	**Share** (%)	**Weight**	**Median**	σ
root	4.54	100.00	10000	-	-
A	1.36	29.99	3000	-	-
A_1	0.27	6.00	600	208.52	12.77
A_2	1.09	23.99	2400	40.77	14.28
B	1.36	29.99	3000	-	-
B_1	0.45	10.00	1000	108.25	17.08
B_2	0.91	19.99	2000	75.15	14.09
C	1.82	40.02	4000	31.58	8.70

Table 6.2: Summary of the singlepath protocol-agnostic transmission

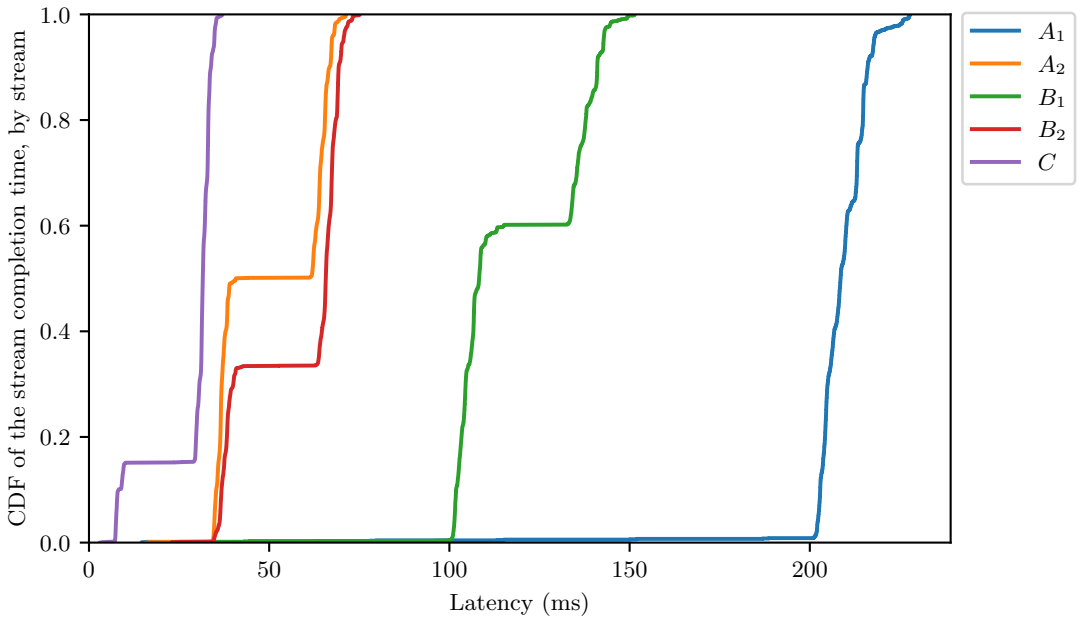

Figure 6.8: CDF plot of the end-to-end latency streaming with a single path

6.1.4 Comparison with HTTP's Extensible Prioritization Scheme

When outlining the Extensible Prioritization Scheme for HTTP [14] used by quiche's default scheduler in Section 2.2, an illustrative accompanying example was provided, which we now execute on the testbed. Table 6.3 re-states the used priorities for the five 100 Mbit bulk transmissions.

The plot in Figure 6.9 reveals a few interesting insights regarding how quiche's implementation in practice deviates from the expected result illustrated in Figure 2.3. By setting, e.g., class C's urgency to 0, the intended outcome was to have a single scheduling cycle that fully transmits C before the server begins transmitting the other streams. Instead, the requests were split into two partial ones, which individually underwent a scheduling cycle. By looking at the source code, we see that the application layer handles this process that is likely intended to avoid starvation, loading massive files into memory, and overwhelming quiche.

Interesting, too, is seeing how B's incremental classes evenly split the bandwidth as they were at the same level of urgency. Our scheduler, in contrast, provides significantly better control regarding how the bandwidth allocation occurs. We cannot have B_1 receiving half of B_2's bandwidth, for instance, as in the hierarchy shown in Figure 6.1.

Lastly, although the HLS paper claims their performance overhead to be "comparable to that of other classful packet schedulers," [17] that is not the case for this first

implementation of the HLS-in-QUIC scheduler. Quiche's server fulfills the requests at approximately 120 Mbit/s, while ours is closer to 66 when unconstrained. The focus of the implementation, however, was not on performance but on fairness — there certainly is room for optimizations in the scheduler's code.

Class	Urgency	Incremental
A_1	1	`false`
A_2	2	`false`
B_1	3	`true`
B_2	3	`true`
C	0	`true`

Table 6.3: Experiment using the Extensible Prioritization Scheme for HTTP

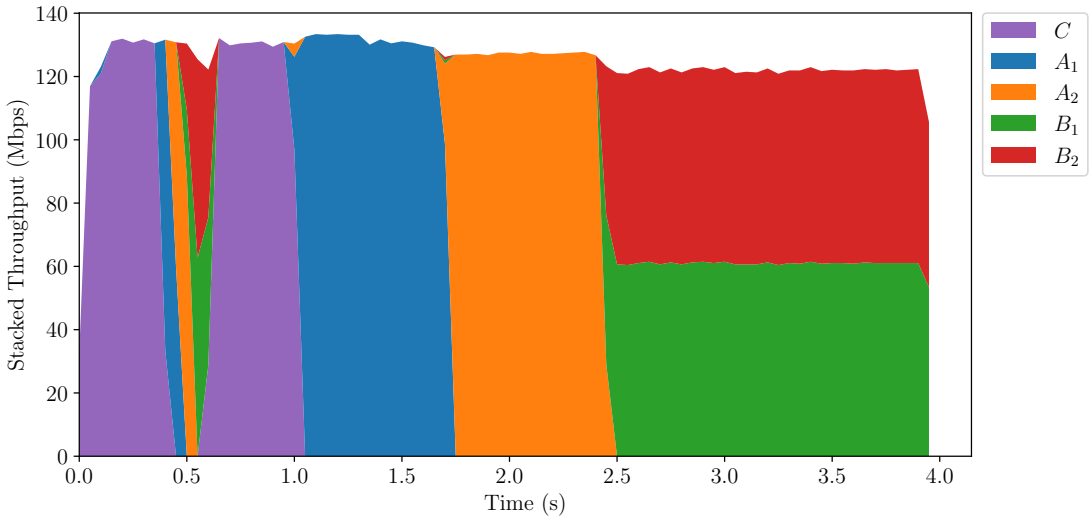

Figure 6.9: Bulk transfer using HTTP/3 extensible priorities

6.2 Multipath Experiments

The command-line interface of quiche's MPQUIC client and server applications only supports assigning a single listening address to the server; the multiple paths come from the same client using two or more source addresses. We followed this same model in Table 6.4 where n clients map to a server, but nothing prevents quiche from supporting n:m paths. As shown in Figure 6.10, we maintain our bridge setup, but in contrast to the singlepath setup, the client splits its dedicated physical link into two separate Virtual Local Area Networks (VLANs). Using Netem, we assign them distinct path characteristics per Table 6.5. The heterogeneous and homogeneous settings come from

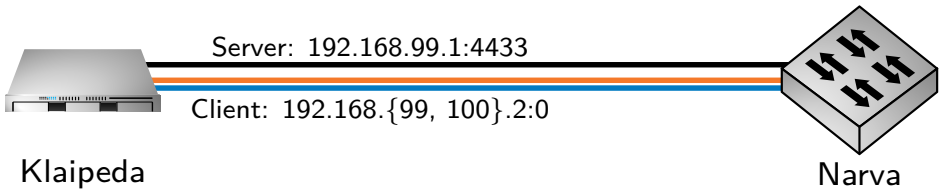

Figure 6.10: Multipath testbed topology

the work by Rabitsch, Hurtig, and Brunstrom [10]. The static IP routes are configured on Klaipeda.

Path	Source	Destination
■	192.168.99.1:4433	192.168.99.2:0
■	192.168.99.1:4433	192.168.100.2:0

Table 6.4: Paths from the server to the client in the multipath topology

Scenario	Path	Bandwidth	RTT
Heterogeneous	■	50 Mbit/s	10 ms
	■	10 Mbit/s	50 ms
Homogeneous	■	50 Mbit/s	10 ms
	■	50 Mbit/s	10 ms

Table 6.5: Path characteristics in the multipath topology

6.2.1 Bulk HTTP/3 Transmission

In this experiment, we re-assess Section 6.1.1 with the multipath topology. The achieved rate refers to the stream's egress throughput from the server while active. The time indicates the duration between when the client sent the `GET` request and when it was fulfilled. The connection is established over the first path.

Under heterogeneity, the LowRTT and SA-ECF schedulers performed similarly. A significant amount of overhead was observed for the 12.5 MB files: each stream carried about 20 MB of data, compared to 13 MB in the singlepath case or with the SA-ECF-based scheduler.

Despite often opting for the slower path, the three highest-weighted classes, A2, B2, and C, completed faster under SA-ECF than LowRTT. However, the total transmission was slower. In contrast, the adapted SA-ECF aggressively waited for the faster path to reopen, almost degenerating into a single-path transmission. Nonetheless, this strategy achieved significantly higher throughput and completion times.

In the homogeneous scenario, the SA-ECF distributed the transmission more equally over both paths, at the expense of their completion time compared to LowRTT in all classes except C. The aggressiveness of the adapted SA-ECF scheduler worked against itself. Despite both paths being similar, it insisted on waiting for a marginally better one, unintentionally sacrificing its throughput. This highlights the delicate balance the SA-ECF strategy attempts to strike and justifies using a hysteresis constant.

In heterogeneous and homogeneous scenarios, the schedulers could observe little stream-awareness. No stream, for instance, fully committed to a specific path while others opted for another. Although that is understandable since the scheduler was written to schedule stream data at each sending opportunity and not full payloads (e.g., "stream 4 on path 1), emerging complex behavior was anticipated but not seen.

Scheduler	**Class**	**Weight**	**Rate** (Mbit/s)	■ (%)	■ (%)	**Time** (ms)
LowRTT	A_1	600	3.02	43.95	56.06	52992
	A_2	2400	4.05	45.15	54.85	36935
	B_1	1000	2.93	43.66	56.34	53415
	B_2	2000	3.55	41.70	58.30	43235
	C	4000	5.77	43.30	56.70	26995
SA-ECF	A_1	600	3.07	40.50	59.50	53382
	A_2	2400	4.33	44.60	55.40	35658
	B_1	1000	3.09	40.18	59.82	53969
	B_2	2000	3.71	44.81	55.19	41764
	C	4000	6.08	44.28	55.72	25680
Adapted SA-ECF	A_1	600	9.05	99.55	0.45	11215
	A_2	2400	12.87	98.15	1.85	8027
	B_1	1000	9.11	99.27	0.73	11276
	B_2	2000	11.37	98.55	1.45	9189
	C	4000	17.65	98.77	1.23	5977

Table 6.6: Scheduler comparison in HTTP/3 bulk transfers with heterogeneous paths

It is pertinent to note the limitations of the current implementation revealed during the experiments of this section. Unlike in the singlepath scenario, the Multipath Fair Stream Scheduler may erratically encounter violations of the HLS invariant. Moreover, quiche's multipath test expects the number of recovery bytes sent per path to be more than half of the useful data, which is not the case with the adapted SA-ECF. It still uses both paths, but the number of recovery bytes sent on the second path is significantly smaller. In Chapter 7, we outline approaches to address these remaining problems.

Scheduler	**Class**	**Weight**	**Rate** (Mbit/s)	■ (%)	■ (%)	**Time** (ms)
LowRTT	A_1	600	13.71	62.50	37.50	7757
	A_2	2400	18.84	62.29	37.71	5528
	B_1	1000	13.77	62.85	37.15	7841
	B_2	2000	16.65	61.64	38.36	6451
	C	4000	25.87	61.48	38.52	4273
SA-ECF	A_1	600	13.67	52.81	47.19	8188
	A_2	2400	18.42	51.50	48.50	5662
	B_1	1000	13.85	53.61	46.39	8227
	B_2	2000	16.08	54.14	45.86	6479
	C	4000	25.40	51.18	48.82	4202
Adapted SA-ECF	A_1	600	11.30	89.55	10.45	11809
	A_2	2400	13.78	93.78	6.22	7679
	B_1	1000	10.82	89.52	10.48	11876
	B_2	2000	12.23	94.26	5.74	9207
	C	4000	18.80	93.84	6.16	5641

Table 6.7: Scheduler comparison in HTTP/3 bulk transfers with homogeneous paths

6.2.2 HTTP/3 Web Traffic

In this experiment, five files totaling 5 MB are served over five HTTP/3 streams. As displayed in Table 6.8, the file sizes differ and conceptually match everyday web page traffic.

Type	**Class**	**File**	**Filesize**	**Weight**
Webpage	root	-	5 MB	10000
Auxiliary	A_1	`style.css`	40 kB	600
	A_2	`script.js`	100 kB	2400
Blobs	B_1	`image.png`	1 MB	1000
	B_2	`video.mp4`	3.8 MB	2000
Content	C	`index.html`	60 kB	4000

Table 6.8: HTTP/3 web traffic pattern experiment

On the heterogeneous topology, we ran the experiment 100 times each for the three schedulers, logging when each stream was completed from the client's vantage point. Figure 6.11 shows the CDF of the runs; those that encountered violations of the HLS invariants were removed. In contrast to the bulk transmission, smaller file sizes enabled the SA-ECF algorithm to estimate stream completion times better, as it modestly improved on LowRTT. However, compared to the SA-ECF variant that more aggressively chooses to wait, it still performed significantly worse. Tinkering with the adapted

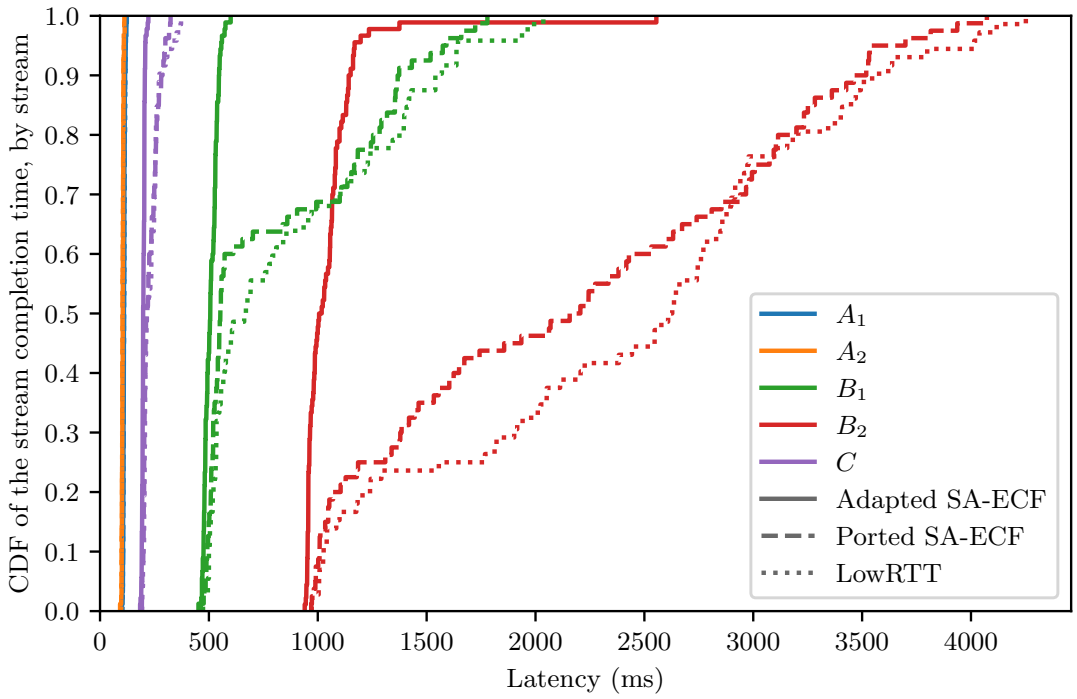

Figure 6.11: Comparison of the SA-ECF and LowRTT schedulers

SA-ECF code revealed that the willingness to wait, and not the consideration of one-way flight delays of the multipath acknowledgments, was responsible for the completion time improvement.

6.3 Reproducibility of the Results

Listing 4 illustrates how to reproduce the raw results of this chapter. The scripts and the scheduler's code are pushed to the GitLab instance of the Leibniz-Rechenzentrum, with the experiment data archived on their storage service. The heart of the HLS implementation is located in its own file[1]; the SA-ECF methods[2] are contained within quiche's `get_send_path_id()` method. Should referenced footnotes not be publicly acessible, access to them can be granted upon request.

[1] `/quiche/-/blob/hls-scheduler/quiche/src/hls_scheduler.rs`

[2] `/quiche/-/blob/hls-scheduler/quiche/src/lib.rs#L8071`

The `setup-node.sh`[1] script resets and installs a fresh Debian Bookworm image on the argument-provided node. Rust version 1.80.0 is not pre-installed, so the script curls the required toolchain. The script then clones several repositories and installs the required dependencies. Our MPQUIC fork[2] implements the Multipath Fair Stream Scheduler. It inits a submodule[3] that contains configuration scripts and client/server applications that use quiche. The HMM module is also present there. A further repository[4] contains Python procedures to generate plots from the transmission's logs. Besides the node name on which to run the installation, the script takes in a specific branch as an argument, too. For example, to run the fair stream scheduler with LowRTT as a path scheduler, `lowrtt` can be passed instead of `hls-scheduler`.

The `run-hls-bridge.sh`[5] script configures a host and bridge node according to the desired topology, given that they differ in the single- and multipath experiments. Once configured, the experiment is started with the client and server on the same host machine. To execute quiche's tests and those of the HMM module, `cargo test` can be run in `/root/quiche`.

The raw results are set to land in `/tmp/HLS`. Besides logs, several plots and textual summaries are automatically generated, providing an initial impression of the transmission's results. Depending on the experiment and the specified hierarchy, the fairness plot script must be manually modified to account for the changed weights, stream IDs, and the fairness bound α. If the Netem configuration defaults from our examples are not desired, they must be configured per Listing 5. Necessary assistance is available for clarifications, as some more complex plots, such as combined CDFs, are produced separately with the raw data of multiple experiments.

---

[1] `/quiche/-/blob/hls-scheduler/setup-node.sh`

[2] `/quiche/-/tree/hls-scheduler`

[3] `/hls/-/tree/hls-scheduler`

[4] `/quiche_vis`

[5] `/hls/-/blob/hls-scheduler/run-hls-bridge.sh`

```
# Allocate nodes involved in the experiment
pos allocations allocate klaipeda narva

# Reset nodes, install dependencies, and clone the repositories
bash setup-node.sh klaipeda <hls-scheduler|lowrtt|multipath|proposal>
bash setup-node.sh narva <hls-scheduler|lowrtt|multipath|proposal>

# Choose one of the available experiments.
# Singlepath:
# - HTTP/3 bulk transmission (5 streams @ 100 Mbit)
bash run-hls-bridge.sh /root/quiche/singlepath-http3.sh

# - HTTP/3 bulk transmission with 50 Mbps and 10ms RTT limit:
bash run-hls-bridge.sh /root/quiche/http3-netem.sh

# - Protocol-agnostic streaming
bash run-hls-bridge.sh /root/quiche/hls/traffic-gen.sh

# - HTTP/3 Extensible Prioritization Scheme (on branch 'multipath')
bash run-hls-bridge.sh /root/quiche/http3-quiche.sh

# Multipath:
# - HTTP/3 bulk transmission
bash run-hls-bridge.sh /root/quiche/http3-bulk.sh multipath

# - HTTP/3 GET of a web page (5 streams, 5 MB total)
bash run-hls-bridge.sh /root/quiche/http3-web.sh multipath
```

Listing 4: Configuration of the topology and execution of the experiments

```
# Configuring Netem on Klaipeda
ip netns exec quic_client tc qdisc del dev enp2s0f0.1 root
ip netns exec quic_client tc qdisc del dev enp2s0f0.2 root

# Path 1: 50 Mbps, 10ms RTT
ip netns exec quic_client tc qdisc add dev enp2s0f0.1 root \
        netem rate 50mbit delay 5ms

# Path 2: 10 Mbps, 50ms RTT
ip netns exec quic_client tc qdisc add dev enp2s0f0.2 root \
        netem rate 10mbit delay 25ms

# Configuring Netem on Narva
tc qdisc del dev enp3s0.1 root
tc qdisc del dev enp3s0.2 root

tc qdisc add dev enp3s0.1 root netem rate 50mbit delay 5ms # Path 1
tc qdisc add dev enp3s0.2 root netem rate 10mbit delay 25ms # Path 2
```

Listing 5: Sample Netem configuration for the multipath topology

Chapter 7

Conclusion

This thesis demonstrates the feasibility of porting the state-of-the-art scheduling strategy the Hierarchical Link Sharing QDisc uses to the transport layer and combining it with stream-aware multipath schedulers. Our implementation in a multipath extension of the QUIC library quiche enables applications to specify a weighted hierarchy at the connection level used to classfully divide traffic. No further coupling to an application protocol or a traffic pattern is present. Classes obtain a byte-granular max-min fair quota of the available throughput, enabling minimum rate guarantees and isolation between classes. By being strategy-proof, the algorithm ensures that classes cannot misrepresent their requests to unfairly obtain higher allocations. The approach offers fine-grained bandwidth control compared to priority-based stream scheduler that quiche ships with.

The component is compatible with the Stream-Aware Earliest Completion First scheduler that decides which stream to include in the next packet and determines the most efficient path for its delivery. The selection is based on how long the stream completion takes if scheduled on a particular path, considering the stream's weight compared to backlogged others. Due to the decoupled architecture of the components, other stream-aware schedulers can easily be integrated if desired. We showed this by comparing the scheduler's performance against quiche's default and one that attempts to consider the differences in MPTCP and MPQUIC data acknowledgments.

The SA-ECF scheduler performed better than the Lowest-RTT-First scheduler in our experiments for web-like HTTP/3 traffic patterns using heterogeneous paths. In bulk file transfers, higher-weighted streams arrived earlier, but the last stream was delayed. However, in both conditions, the slow path was still often chosen. An SA-ECF variant

that insists on the faster path choice more strongly significantly reduced stream completion times in heterogeneous environments, but its performance degraded in homogeneous scenarios. These results show that more research into novel MPQUIC scheduling strategies is required that do not necessarily build atop MPTCP approaches. Table 7.1 gives an overview of the requirements fulfilled by the scheduler, as elicited in Table 3.1.

Requirement	**Status**
Multipath scheduling	✓
Transparent to the application	✓
Stream-aware scheduling	✓
Minimum guarantees	✓
Strategy-proof fairness	✓
Hierarchical bandwidth sharing	✓
React to network condition changes	○

TABLE 7.1: Fulfilled requirements of the Multipath Fair Stream Scheduler

Regarding future work, a few remaining issues need to be addressed. When it came to artificially constraining the bandwidth with Netem, the singlepath bulk transmission showed a few odd spikes despite never violating the HLS invariant from Formula 5.4. In multipath scenarios, such violations occurred sporadically. The issue is likely related to how the round-size quantum Q^* is selected. We followed HLS's implementation, which dynamically adjusts the quantum to ensure at least one packet is sent out per round. However, that may not be necessary since we deal with single bytes in stream frames instead of packets. A better approach may be to base the scheduler's quantum on the send quantum provided by quiche, which is the maximum packet burst size given by the congestion control. The scheduler could reset after the main and surplus rounds, initializing anew with the updated Q^*. That would enable the use of relative weights — as shown in the HMM module — unlike the current absolute byte guarantees.

Another approach could be to have one HLS scheduler per path, especially if a stream-aware scheduler is used to place entire streams on a certain path. For that to be realistic, however, performance optimizations need to happen. This work could occur as part of student theses that could also investigate the accuracy of the `bytesUntilCompletion` method, measuring the number of out-of-order packets and adding support for QUIC datagrams. Finally, another question is how to utilize paths deemed too slow in heterogeneous environments. While they may be too slow for stream transmission, they may be helpful for proactive loss recovery mechanisms such as forward error correction. Running such experiments would need more advanced traffic generators and topologies than we had, using several testbed nodes synchronized with the Precision Time Protocol, for instance, rather than our bridge setup.

Chapter A

Appendix

A.1 Acronyms

ADU	Application Data Unit
API	Application Programming Interface
CBQ	Class-Based Queueing
CDF	Cumulative Distribution Function
CPU	Central Processing Unit
CSS	Cascading Style Sheets
ECF	Earliest Completion First
HBES	Head-of-line Blocking Eliminating Scheduler
HLS	Hierarchical Link Sharing
HMM	Hierarchical Max-Min
HTB	Hierarchical Token Bucket
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HoLB	Head-of-Line Blocking
ID	identifier
IETF	Internet Engineering Task Force
IP	Internet Protocol
JS	JavaScript
KPI	Key Performance Indicator
LCP	Largest Contentful Paint
LowRTT	Lowest-RTT-First
MPQUIC	Multipath QUIC

MPTCP	Multipath TCP
OFO	out-of-order
OSI	Open Systems Interconnection
PSM	Priority-Based Stream Manager
QDisc	Queueing Discipline
RFC	Request for Comments
RR	Round-Robin
RTT	round-trip-time
SA-ECF	Stream-Aware Earliest Completion First
SAPS	Stream-Aware Arrival-Time-Based Path Selector
SRPT	Shortest Remaining Processing Time
TCP	Transmission Control Protocol
TUM	Technical University of Munich
UDP	User Datagram Protocol
VLAN	Virtual Local Area Network
WLAN	Wireless Local Area Network
WRR	Weighted Round-Robin

IETF's QUIC is *not* short for "Quick UDP Internet Connections." [1]

Bibliography

[1] J. Iyengar and M. Thomson, *QUIC: A UDP-based multiplexed and secure transport*, RFC 9000, May 2021. DOI: 10.17487/RFC9000. [Online]. Available: https://www.rfc-editor.org/info/rfc9000.

[2] Y. Liu, Y. Ma, Q. D. Coninck, O. Bonaventure, C. Huitema, and M. Kühlewind, "Multipath extension for QUIC", Internet Engineering Task Force, Internet-Draft draft-ietf-quic-multipath-08, May 2024, Work in Progress, 35 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-multipath/08/.

[3] V. A. Vu and B. Walker, "On the latency of multipath-QUIC in real-time applications", in *2020 16th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, 2020, pp. 1–7. DOI: 10.1109/WiMob50308.2020.9253402.

[4] O. Bonaventure, M. Piraux, Q. D. Coninck, M. Baerts, C. Paasch, and M. Amend, "Multipath schedulers", Internet Engineering Task Force, Internet-Draft draft-bonaventure-iccrg-schedulers-02, Oct. 2021, Work in Progress, 14 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-bonaventure-iccrg-schedulers/02/.

[5] Y. Xing, K. Xue, Y. Zhang, J. Han, J. Li, D. S. L. Wei, R. Li, Q. Sun, and J. Lu, "A stream-aware MPQUIC scheduler for HTTP traffic in mobile networks", *IEEE Transactions on Wireless Communications*, vol. 22, no. 4, pp. 2775–2788, 2023. DOI: 10.1109/TWC.2022.3213638.

[6] M. Bishop, *HTTP/3*, RFC 9114, Jun. 2022. DOI: 10.17487/RFC9114. [Online]. Available: https://www.rfc-editor.org/info/rfc9114.

[7] B. Y. L. Kimura, D. C. S. F. Lima, and A. A. F. Loureiro, "Packet scheduling in multipath TCP: Fundamentals, lessons, and opportunities", *IEEE Systems Journal*, vol. 15, no. 1, pp. 1445–1457, 2021. DOI: 10.1109/JSYST.2020.2965471.

[8] B. Jonglez, M. Heusse, and B. Gaujal, "SRPT-ECF: Challenging round-robin for stream-aware multipath scheduling", in *2020 IFIP Networking Conference,*

Networking 2020, Paris, France, June 22-26, 2020, IEEE, 2020, pp. 719–724. [Online]. Available: https://ieeexplore.ieee.org/document/9142713.

[9] Q. D. Coninck, *Quiche*, https://github.com/qdeconinck/quiche/tree/multipath. (visited on 08/09/2024).

[10] A. Rabitsch, P. Hurtig, and A. Brunström, "A stream-aware multipath QUIC scheduler for heterogeneous paths", in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC, EPIQ@CoNEXT 2018, Heraklion, Greece, December 4, 2018*, ACM, 2018, pp. 29–35. DOI: 10.1145/3284850.3284855. [Online]. Available: https://doi.org/10.1145/3284850.3284855.

[11] 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*, ser. CoNEXT '17, Incheon, Republic of Korea: Association for Computing Machinery, 2017, 147–159, ISBN: 9781450354226. DOI: 10.1145/3143361.3143376. [Online]. Available: https://doi.org/10.1145/3143361.3143376.

[12] Cloudflare, *Quiche documentation*, https://docs.quic.tech/quiche/index.html, n.d. (visited on 08/09/2024).

[13] L. Pardue and A. van der Mandele, *Introducing HTTP/3 prioritization*, https://blog.cloudflare.com/better-http-3-prioritization-for-a-faster-web/, Cloudflare, 2023. (visited on 08/09/2024).

[14] K. Oku and L. Pardue, *Extensible prioritization scheme for HTTP*, RFC 9218, Jun. 2022. DOI: 10.17487/RFC9218. [Online]. Available: https://www.rfc-editor.org/info/rfc9218.

[15] T. Pauly, *Call for adoption: Extensible prioritization scheme for HTTP*, https://lists.w3.org/Archives/Public/ietf-http-wg/2019OctDec/0181.html, 2019. (visited on 08/09/2024).

[16] Y. Weiss and N. P. Moreno, *Largest contentful paint*, https://www.w3.org/TR/largest-contentful-paint/, W3C Working Draft, 15 January 2024, 2024. (visited on 08/09/2024).

[17] N. Luangsomboon and J. Liebeherr, "A round-robin packet scheduler for hierarchical max-min fairness", *CoRR*, vol. abs/2108.09864, 2021. arXiv: 2108.09864. [Online]. Available: https://arxiv.org/abs/2108.09864.

[18] I. Marsic, *Computer Networks: Performance and Quality of Service*. New Brunswick, NJ: Rutgers University, 2013. [Online]. Available: http://eceweb1.rutgers.edu/~marsic/.

[19] N. Luangsomboon and J. Liebeherr, "HLS: A packet scheduler for hierarchical fairness", in *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, 2021, pp. 1–11. DOI: 10.1109/ICNP52444.2021.9651972.

[20] Lantua, *Hierarchical link sharing*, 2021. [Online]. Available: https://github.com/lantua/HLS (visited on 08/09/2024).

[21] M. Fritz, "State of the art assessment of multipath QUIC", Bewertung des aktuellen Stands von Multipath QUIC, Supervisor: Prof. Dr.-Ing. Georg Carle, Advisor: Killian Holzinger, Lion Steger, Marcel Kempf, Master's Thesis, Technical University of Munich, School of Computation, Information, and Technology, Informatics, 2024.

[22] Q. De Coninck and O. Bonaventure, "Multipath QUIC: Design and evaluation", in *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '17, Incheon, Republic of Korea: Association for Computing Machinery, 2017, 160–166, ISBN: 9781450354226. DOI: 10.1145/3143361.3143370. [Online]. Available: https://doi.org/10.1145/3143361.3143370.

[23] M. Bünstorf, "Msquic – a high-speed QUIC implementation", in *Proceedings of the Seminar Innovative Internet Technologies and Mobile Communications (IITM), Summer Semester 2023*, G. Carle, S. Günther, B. Jaeger, and L. Seidlitz, Eds., ser. Network Architectures and Services (NET), vol. NET-2023-11-1, Munich, Germany: Chair of Network Architectures, Services, School of Computation, Information, and Technology, Technical University of Munich, Dec. 2023, pp. 1–5. DOI: 10.2313/NET-2023-11-1_01.

[24] T. Pauly, E. Kinnear, and D. Schinazi, *An unreliable datagram extension to QUIC*, RFC 9221, Mar. 2022. DOI: 10.17487/RFC9221. [Online]. Available: https://www.rfc-editor.org/info/rfc9221.

[25] M. Haden, "I8-testbed: Introduction", in *Proceedings of the Seminar Innovative Internet Technologies and Mobile Communications (IITM), Summer Semester 2020*, G. Carle, S. Günther, and B. Jaeger, Eds., ser. Network Architectures and Services (NET), vol. NET-2020-11-1, Munich, Germany: Chair of Network Architectures and Services, Department of Computer Science, Technical University of Munich, Nov. 2020, pp. 61–65. DOI: 10.2313/NET-2020-11-1_12.