Forward Error Correction and Weighted Hierarchical Fair Multiplexing for HTTP/3 over QUIC

Kilian Holzinger, Daniel Petri,
Stefan Lachnit, Marcel Kempf, Henning Stubbe, Sebastian Gallenmüller, Stephan Günther, Georg Carle
Technical University of Munich, Germany
{holzinger, petriroc, lachnit, kempf, stubbe, gallenmu, guenther, carle}@net.in.tum.de

***Abstract*—Web applications are ubiquitous and increasingly use HTTP/3. Their performance is affected by the underlying QUIC transport protocol. An important performance metric is the transmission delay impacted by the standardized loss recovery and resource prioritization. To improve the robustness against packet loss, we extend QUIC's recovery mechanism by the convolutional Forward Error Correction scheme Tetrys. For better control over the order of sent data, we use a round-robin scheduler that provably ensures hierarchical max-min fairness of the multiplexed streams at a byte-granular level. We extend its functionality to support strict priorities within a scheduling tree of weighted classes, integrating it into the Extensible Prioritization Scheme for HTTP/3. Measurements of a prototype implementation demonstrate that transmission delays improve under common web workloads and that the scheduler delivers important assets earlier with the newly specified parameters.**

***Index Terms*—QUIC, Forward Error Correction, Multiplexing**

I. Introduction

HTTP/3, which uses QUIC as a transport protocol, is increasingly adopted, e. g. in modern real-time multimedia web applications. For such use cases, transmission delays are a crucial performance metric. Among others, they are impacted by network characteristics and the transport protocol as well as its implementation. Factors, such as network paths properties, are beyond the control of endpoints. In contrast, the upper layers, i. e. the transport and application protocols, can be changed. This flexibility has been one of the core rationales leading to QUIC's design [1]. The standardized reliability mechanism and the scheduling of multiplexed streams are a source of delay [2], [3]. In this work, we propose additional approaches for loss recovery and stream prioritization to demonstrate benefits for common web application workloads.

QUIC offers parallel streams to avoid head-of-line (HOL) blocking. HTTP/3 uses the Extensible Prioritization Scheme (EPS) to send priority signals. Suitable EPS parameters indicate web browsers to increase parallelism. However, in practice, due to limited flexibility of the resource prioritization, QUIC cannot yield significant advantages in parallel stream scenarios [3], [4]. Though parallelism has proven beneficial in scenarios with high random loss, bursty loss, affecting several streams at once, has led to detrimental effects [5]. Related work suggests that there is no prioritization scheme that performs best in all use cases and that new priority metadata need to be discussed, such as clients recommending their preferred order of information delivery [6] and a more granular control over the prioritization [7].

Forward Error Correction (FEC) is an alternative loss recovery method. It transmits redundant information, so receivers can restore original data, as long as enough linear independent information has arrived, avoiding costly retransmissions. FEC was considered early in QUIC's development but has not been standardized yet. However, there is ongoing scientific discussion to include FEC recovery mechanisms within QUIC over the past years [8]–[12].

In this paper, we show that Application Data Unit (ADU) transmission times for HTTP/3 over QUIC can be improved by introducing new priority signals, a FEC-based reliability mechanism, and a flexible stream scheduler. We use an experimental protocol extension for the FEC-based reliability mechanism. Through additional EPS parameters, endpoints indicate reliability properties and the desired position within a hierarchical priority tree for each HTTP/3 request. A mixture of strict priority and weighted round-robin scheduling enables simultaneous sequential, incremental, and weighted parallel transmission of stream data while remaining fallback-safe to EPS. We propose, implement, and evaluate an according scheduling algorithm with proven max-min fairness guarantees. The approach is assessed using web application workloads and measurements of a prototype implementation in an emulated network environment. There, the results show improved delays for non-incremental and incremental transfers under lossy path conditions and earlier completion of relevant resources through more flexible stream prioritization. The source code of our prototype is available online [13].

The remainder of this work is structured as follows: Section II describes the FEC coding scheme and its integration into QUIC. Afterward, Section III shows the integration of the hierarchical weighted round-robin scheduler into the stream multiplexing. In Section IV, new HTTP/3 priority signals are introduced that enable to leverage the flexibility of the new mechanisms in HTTP/3 applications. Section V assesses the impact of the improved recovery and scheduling on typical web workloads. Finally, before the concluding Section VII, related work is summarized in Section VI.

II. Forward Error Correction

Over the past years, several studies have experimented with FEC in QUIC, investigating different coding schemes. A thor-

ISBN 978-3-903176-72-0

ough overview of related works is provided in Section VI-A. A recent publication on that topic, QUIRL, identified a coding scheme called Tetrys as the most suitable for use within the QUIC protocol [12]. Thus, our work uses a Tetrys coding scheme, and derives the protocol modification from the RFC draft published by the QUIRL authors [14].

This section provides background on the Tetrys coding scheme and its integration into QUIC. We explore suitable scheduling of redundant information for common HTTP/3 web workloads. Different approaches for bulk transfers and incremental or streaming ADUs are needed. In this paper, we use the terminology suggested by the Coding for efficient NetWork Communications Research Group (NWCRG) [15].

A. Tetrys Coding Scheme

Tetrys is a systematic rateless convolutional random linear erasure coding scheme for unicast communication [16]. The concept was published as an experimental RFC by the IRTF, too [17]. It includes a back channel for symbol acknowledgments from the decoder to the encoder. At the encoder, it uses a dynamic encoding buffer which holds unacknowledged Source Symbols, referred to as the encoding window. Interspersed with the Source Symbols (SSs), Repair Symbols (RSs) are transmitted. They are computed, using Galois Arithmetic, as a linear combination of the encoding window with coefficients being randomly drawn from a Galois Field. To avoid transmitting the coefficients, the authors advise to transmit the seed for a pseudo-random number generator and use it to reconstruct them at the decoder. The decoder keeps arriving SSs and RSs in a decoding window. To reconstruct missing symbols, the decoding window is arranged into a matrix of coefficients and the system of linear equations is solved with Gaussian elimination. The original authors provide an extensive simulation-based study on the behavior of the coding scheme in a wide range of network path properties. They find that Tetrys can achieve loss recovery independently from Round Trip Time (RTT) and is robust against bursty loss—a common loss pattern on Internet paths [16].

B. Implementation of Tetrys

In this work, we implemented the Tetrys scheme as a Rust module, split into encoder and decoder functionality. For necessary Galois Field arithmetic operations, we use the hardware-accelerated open-source library libmoepgf [18] and bindgen[0] to generate bindings for Rust. We choose the field $GF(256)$, as it matches well with available vector instructions and offers a sufficiently large coefficient space. The SIMD instructions expect aligned data structures so we use aligned_vec[1] for memory-aligned byte vectors. In contrast to the original Tetrys proposal, for simplicity, no selective symbol acknowledgments are supported. The coefficients are derived from a seeded Pcg64Mcg pseudorandom number generator.

[0]https://github.com/rust-lang/rust-bindgen

[1]https://docs.rs/aligned-vec/latest/aligned_vec/

C. Integration into QUIC

The used FEC extension is compliant with QUIC and based on a standards draft by the authors of QUIRL [14]. New transport parameters are used, so only compatible endpoints will use the FEC functionality. In contrast to the protocol draft [14], several modifications, based on requirements of this work and experiences during development, were made. FEC protected frame types, such as `STREAM` frames, are encapsulated in `SOURCE_SYMBOL` frame headers. We extend those, like all additional frame types, with a `fec_session` field, to enable multiple distinct FEC contexts within a single connection to reflect differentiated reliability requirements. Also, we add a length field and a Source Symbol identifier (SID), which is incremented for each sent frame and is used to detect missing SSs at the decoder and is referenced by RSs headers. The `REPAIR` frame header includes the smallest and largest SID to enable reconstruction of the encoding window at the receiver. The seed used to derive the coefficient vector, in our implementation, equals the largest SID plus an additional seed offset field. So generating multiple RSs from the same encoding window yields different linear combinations. We add a lower bound to the `SYMBOL_ACK` frame, telling the decoder the current lowest SID in the encoding window. This proved to be a necessary information to keep the decoding window small in situations where not many RSs are sent. We follow the recommendation to treat reconstructed information as lost in the congestion controller [19] and implemented our changes in the Cloudflare quiche library[2].

D. Scheduling of Repair Symbols

RS scheduling decides when to send repair information. In general, SSs are preferred over RSs, to avoid additional delay, and to offer full path capacity if required by the application. We anticipate that not all HTTP transactions require FEC.

1) Transmission delay and value of information: Web applications create ADUs of different sizes and with varying frequency. Some ADUs, e. g. file downloads, cannot be used partially. They are not useful until they are fully available at the destination. Thus, the time to completion (TTC), measured from request to a completely received reply, is a suitable metric to minimize. Other ADUs can be used incrementally. For example, many web browsers can parse and render partial HTML documents. This is reflected in the `i` parameter of the EPS. Similarly, streaming data, such as real-time multimedia content, transfers smaller ADUs in quick succession. In these cases, a suitable metric to minimize is the byte-wise one-way delay (OWD). It measures the time from when a particular offset position in the data stream is created at the sending application to the time of arrival at the receiving application.

2) Repair delay tolerance (RDT): We introduce the RDT as a parameter to the RS scheduler. It expresses the additional tolerable waiting time for all data to arrive when loss recovery is necessary. In general, neglecting processing overhead, coded repair information is necessary only if the RDT is smaller

[2]https://github.com/cloudflare/quiche/

than the time needed for retransmission. The QUIC loss detection timeout is $\frac{9}{8}RTT$ [20]. Assuming a symmetric path, a retransmitted stream segment travels $\frac{1}{2}RTT$. This makes automatic repeat request (ARQ) based recovery sufficient if the recovery delay tolerance is greater than $\frac{13}{8}RTT$. In other cases, FEC based recovery can improve the end-to-end OWD. Besides deciding whether FEC-based recovery is beneficial, the RDT parameter expresses a certain flexibility to the timing of the recovery, which can be used to improve robustness. Many loss effects, e. g. on Internet paths, are bursty, typically caused by short overloads in the network. The RDT allows to separate the transmission of RSs from the SSs, reducing the chance that they are affected by the same burst loss event. Thus, we trade increased repair delay with an increased chance that either the SSs or corresponding RSs arrive successfully.

3) Burst loss tolerance (BLT): The second parameter for RS scheduling is the BLT. It describes how many RSs should be sent on each occasion, likely allowing the decoder to recover from that amount of lost SSs.

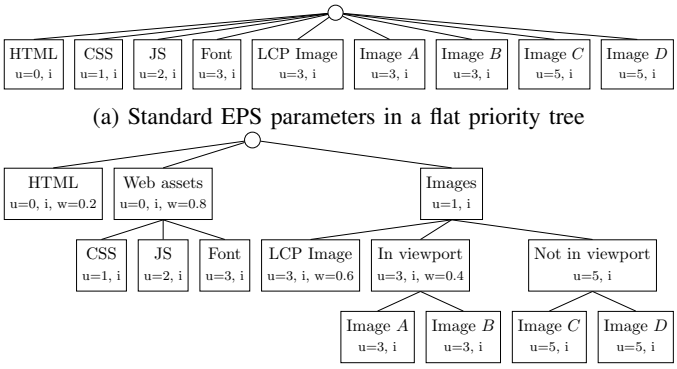
4) Bulk transfers: We optimize the TTC of non-incremental assets by sending RSs in application limited phases. Those are initiated either by a `FIN` stream flag or when the stream send buffer is flushed, i. e. all stream data have been sent, and the application does not provide new data. When detecting loss, the standard QUIC recovery mechanism is used. Thus, during large transfers, there is little additional overhead in the form of SS frame headers. Losses of the last stream segment in flight are protected by RSs and, thus, download times can potentially be reduced by avoiding retransmissions. The approach works similar to previous works [11], [12].

5) Incremental assets: Incremental or streaming assets should be optimized for the byte-wise OWD. When an incremental stream sends an SS, a timer is started waiting for the RDT. After the timer has expired and no `SYMBOL_ACK` for that SS has arrived, RSs are sent.

6) Adaption of FEC parameters: In this work, we do not provide algorithms to adapt the BLT parameter dynamically. Endpoints may request changes to the coding parameters with `PRIORITY_UPDATE` frames based on some estimation, heuristic, or algorithm. We refer to related work, that provides approaches that could be considered [12], [21], [22].

III. QUIC Multiplex Scheduling with Weighted Hierarchical Max-Min Fair Round-Robin

The standard prioritization scheme for QUIC is the EPS, based on strict priorities. With only two parameters (u, i), its flexibility is limited [6], [7]. This section describes a new fallback-safe approach which provides more control what is sent when, motivated by the need of modern web applications. It adds capabilities to arrange streams in hierarchical trees, fitting to the hierarchical structure of web documents. and introduces weighted round-robin scheduling. It enables more parallelism, avoiding HOL blocking, and incorporates the state-of-the-art algorithm Hierarchical Link Sharing (HLS) offering max-min fairness guarantees.



(a) Standard EPS parameters in a flat priority tree

(b) Experimental EPS parameters in a priority tree

Fig. 1: Example scheduling scenario

A. The Hierarchical Link Sharing Packet Scheduler

HLS is a classful packet scheduler for granular and hierarchical traffic control. Designed as a weighted round-robin scheduler, it is optimized for high throughputs. While the edges of an HTTP/2 priority tree imply dependencies between vertices [23], HLS classes fairly isolate how link bandwidth is distributed over multiple aggregation levels [24]. Only the leaves of the hierarchy are assigned traffic. Given a constrained resource at the hierarchy's root, such as an Ethernet link with limited capacity (e. g. 1 Gbit/s), HLS applies the max-min fairness to each parent such that they can subdivide the capacity made available to them using computed fair shares. This hierarchical max-min (HMM) fair allocation accounts for the children's relative weights and how much of the resource they request (e. g. a rate of 200 Mbit/s). Since children of the same parent allocate bandwidth relative to their weights, starvation is prevented. In fact, classes have a minimum rate guarantee derived from the root's capacity and the weighted hierarchy. The guarantee is only exceeded when other classes request less bandwidth than they are entitled to, freeing capacity that the HMM principle re-distributes to classes with unmet demands. The process is strategy-proof. Misrepresented requests cannot game allocations to obtain an advantage.

B. EPS-based HMM scheduling for QUIC

Our scheduler differs from the HLS design in these aspects:

1) Byte-granular stream scheduling: Implemented as a qdisc for the Linux kernel, HLS schedules packets. Classes keep track of a balance in bytes that can be spent to emit traffic; a leaf class only transmits if it has enough balance to cover the packet size at the head of the queue. When porting the approach to QUIC, we have no such limitation: each leaf class is a QUIC stream whose balance is consumed by the length of QUIC frames.

2) Relative weights: The HLS authors use the terms weight w and guarantee g interchangeably, as they show how ratios at the sibling level can be converted into rate guarantees when the link capacity is known. The analog for the link capacity in our scenario is the connection bandwidth, which is not necessarily known, e. g. by the web developers. For ease of use, `exp_w` is specified relative to its siblings with a default value of one.

Internally, it is recomputed into a weight ranging from 0 to 1 for the root, from which the guarantees and round robin quantum are computed. By doing so, we aim to make our prioritization scheme more intuitive, by resembling HTTP/2 weights, and abstracting away the scheduler's intricacies.

3) Round size: A scheduling round visits a set of leaf classes $\mathcal{L}_s$ in a round-robin fashion. The determination of the schedule $\mathcal{L}_s$ from a tree with leaf classes $\mathcal{L}$ is detailed in the next paragraph. When a leaf is visited for the first time, its parent calculates a fair share used to give children balance accounting for their weights. The round-robin is advanced either once the stream has no available balance left or it backlogs no more data and returns the excess balance to the parent for redistribution. Since the determination of the fair shares is a recursive process, the root node is eventually reached. The guarantee in bytes g_{root} that the root can distribute is the round robin's quantum Q^*. The larger Q^*, the longer the duration of a scheduling round. HLS chooses the quantum such that at least one packet can provably be sent in a round, which considers the maximum packet size $L^{\max}$ of a class and prevents the scheduler from iterating without transmission. Our ported implementation, in contrast, emits as long as the balance is positive since a single byte can be sent in a QUIC stream frame albeit with significant overhead. To prevent that, we aim for a Q^* that still gives the stream with the lowest absolute weight enough balance to fill the remaining space of a frame's payload. We set $L^{\max}$ to 1350 B to approximate that value, which is the maximum outgoing UDP payload size of our quiche client. Our heuristic is: $g_{\text{root}} = Q^* = L^{\max} \cdot (1 + \sum_{i \in \mathcal{L}_s} \frac{w_i}{w_{\min}})$

4) Incorporation of strict priorities: We can think of standard EPS scheduling as a special case of HLS where all classes are incremental, have the same weight, and urgency. The incorporation of weights can be achieved using the HLS scheduling strategy alone; it is the integration of strict priorities inside classes that requires a new scheduling strategy. For this more general stream scheduler, we first determine which classes should be scheduled next using the hierarchy and active streams as input. The standard HLS scheduler takes the set of active leaves $\mathcal{L}_{ac} \subseteq \mathcal{L}$ as input for scheduling decisions. These are the streams ready to emit buffered data. Similarly, an internal class in $\mathcal{I}_{ac}$ is active if any children are. To account for the EPS strict priority signaling, we filter $\mathcal{L}_{ac}$ further. Our scheduling strategy traverses the tree starting at the root using a modified Breadth-First-Search (BFS) algorithm to compute the schedule $\mathcal{L}_s \subseteq L_{ac}$. The BFS queue B is initialized with the root. While there is a first element b_1 to dequeue, we reveal its n active children classes v. We combine them into a tuple of vertices V, sorted by EPS priority: $V \coloneqq \langle v_1, \ldots, v_n | v \in \text{child}(b_1) \cap (\mathcal{L}_{ac} \cup \mathcal{I}_{ac}), v_1 \geq \ldots \geq v_n \rangle$. Note that the $\geq$ relation refers to $\geq_{EPS}$, i. e. the first element in V has the highest priority as long as it is not empty. For a total order enabling class comparisons, we use internal class IDs that reflect the request sequence instead of solely relying on the stream ID. With the function $u(v)$ returning the urgency of a class, we then only retain classes at the highest contained urgency level: $V \coloneqq \langle v_k \mid u(v_k) = u(v_1) \rangle$ Elements in V are processed in the following manner, breaking after the first non-incremental class: $\begin{cases} v_k \in \mathcal{L}_{ac} & \implies \mathcal{L}_s \coloneqq \mathcal{L}_s \cup \{v_k\} \\ v_k \notin \mathcal{L}_{ac} & \implies \text{enqueue}(v_k, B) \end{cases}$

5) Dynamic hierarchy: HLS uses a static, pre-defined hierarchy for scheduling. In dynamic web environments, however, streams are added, removed, and re-prioritized ad hoc. We build the hierarchy as HTTP/3 requests arrive, considering new streams for scheduling once the current round completes. Since high-priority streams should transmit as soon as possible, we try to keep round sizes as small as possible. A preemptive solution could expense HMM fairness for the round instead. We reprioritize on-the-fly, which can come with a tractable fairness penalty for the round. Streams that finish are idly kept in the hierarchy until the round's completion. Hierarchies are stored per endpoint at the connection level, e. g. Figure 1b illustrates the state of a server-side weighted class hierarchy populated with EPS parameters when all assets of a sample webpage have been requested but not yet fully delivered. Each endpoint stores their own hierarchy.

Parameter	Type	Description
`exp_p`	Inner List	List of parents in ascending order
`exp_b`	Integer	Burst loss tolerance (BLT)
`exp_d`	Integer	Repair delay tolerance (RDT) in ms
`exp_w`	Decimal	Hierarchical max-min (HMM) relative weight

TABLE I: Additional experimental EPS parameters

IV. New HTTP/3 Priority Parameters

The following section describes EPS and priority signals that an HTTP/3 endpoint can send. As the name suggests, it allows to extend the parameter set used for resource scheduling. Our implementation allows to express desired FEC configurations and HMM parameters in HTTP/3 requests. We use the prefix `exp_` to indicate the experimental nature of our new parameters (recommended by RFC [25]).

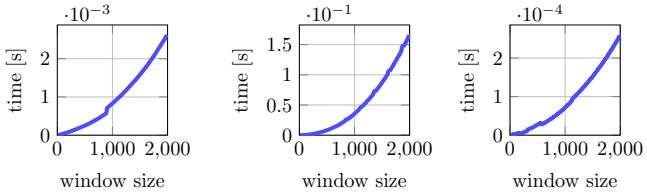
A. Extensible Prioritization Scheme (EPS) for HTTP/3

The EPS indicates request priorities by two parameters. The integer urgency parameter u indicates a strict priority, with 0 as the highest and 7 as the lowest priority. The incremental parameter i is a boolean. Incremental requests of the same urgency are transmitted in a equally weighted round-robin manner. In case non-incremental requests of the same urgency need to be sent, the scheduling follows the request order [25]. The priority parameter is a string in the Structured Field Values (SVF) format [26]. It can be included in the `priority` field of an HTTP/3 request header. An alternative way to indicate the priority is the `PRIORIY_UPDATE` HTTP/3 frame. The latter can be used to reprioritize a request already sent.

B. Experimental Parameters for FEC and HMM Fairness

Table I lists the new parameters which can be used in addition to the u and i parameters. The weight parameter is relative and is normalized by the scheduler.

The `exp_p` parameters is an Inner List, whose elements, in accordance with SVF, are separated by white spaces. The



(a) Encoding time for a single repair symbol

(b) Decoding time with 5% of source symbol loss at the begin of the window

(c) Decoding time with 5% of source symbol loss at the end of the window

Fig. 2: Benchmark results for encoding and decoding

elements of the list are strings with SVF parameters joined by semicolons. The string acts as an identifier for the parent node, used to match different requests within the same connection to the tree structure. Initially the tree is empty, just containing the root node and is populated by subsequent requests. If requests have disagreeing priority values of internal nodes, then the latest request takes precedence. The FEC parameters `exp_b` and `exp_d` are only relevant for leaf nodes and ignored by internal nodes. Requests without an `exp_p` parameter are placed under the root—therefore remaining fallback-safe. Since endpoints may not support and, thus, ignore priority information beyond the urgency parameter and the incremental flag, sensible fallback-safe defaults should be supplied [25]. As an example, a valid EPS priority field value of *Image A* in Figure 1b could look like: `priority = u=3, i, exp_b=3, exp_d=80, exp_p=("ivp";u=3;i;exp_w=0.4 "img";u=1)`

V. Evaluation

A. Forward Error Correction Coding Performance

Micro benchmarks of our modified QUIC library analyze the performance of the Tetrys implementation. A key factor is the size of the encoding and decoding windows. We use the Criterion.rs[3] benchmarking library and define several benchmark groups, investigating window sizes between 10 and 2000 SSs. The encoding window size corresponds to the number of in-flight symbols. To help setting the presented numbers into relation, the following datapoint is provided: with MTU sized SSs, an RTT of 100 ms, and a throughput of 100 Mbit/s roughly 833 symbols would be in flight. Each benchmark is split into an initialization phase which is excluded from the execution time measurement, an phase where necessary data structures are initialized according to the input parameters, and a (cache) warmup phase whose output is discarded. Subsequently, the actual measurements are performed. The benchmarking library repeats the measurements until stable results are achieved. We report the mean times of these results. 0

1) Encoding: The benchmark of the encoding process uses an encoding window as an initialized data structure. This window consists of 1500 B vectors with random data. The benchmarked function generates a single RS from the encoding window. Figure 2a shows the average encoding time for the benchmark results on an Intel Xeon D-1518 CPU. For the window size 1000, the encoding time stays well below 1 ms, for 2000 below 3 ms.

[3]https://bheisler.github.io/criterion.rs/book/index.html

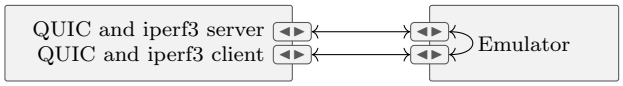


Fig. 3: Experiment setup for the evaluation

2) Decoding: The variable in the decoding benchmark group is the decoding window size. The coding library is initialized by populating an encoding window with random SSs, and then providing 95% of the SSs to the decoder along with 5% RSs. We distinguish between early loss, close to the front of the window, and late loss, during the last few SSs of the window. The results show a significant difference. For early losses, at a window size of 1000, it takes around 35 ms to restore all lost symbols, as depicted in Figure 2b. In contrast, for late losses at the same window size, it takes less than 0.1 ms (cf. Figure 2c).

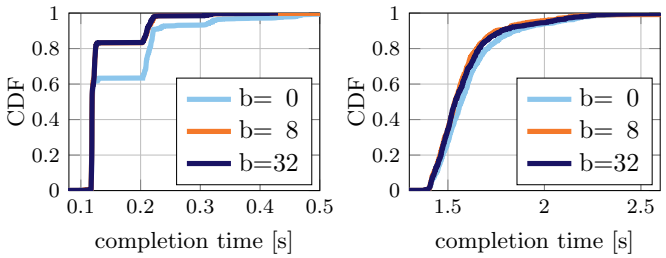
3) Discussion: The presented encoding and decoding times are sufficient for many web workloads. Decoding with early losses has increased runtime, requiring more operations during the Gaussian elimination. We think that our decoding solver can be improved, e. g. by reducing the number of operations performed on Symbol data. Still, we deem the performance of the Tetrys implementation as viable for many web scenarios as we expect only a fraction of requests needing FEC protection.

B. Transmission with Forward Error Correction

In this subsection, experiments of the prototype FEC implementation are evaluated. First, a measurement campaign to assess the transfer completion time of non-incremental web assets is presented. Then, the observed OWD during the transmission of incremental web workloads are studied in an experimental setting.

1) Setup: To obtain the results in a reproducible environment, we created an experimental setup consisting of two nodes with two network interfaces each, depicted in Figure 3. Both machines run Linux (Debian Bookworm). The *Emulator* node uses a Linux software bridge to connect the two interfaces. We emulate network path properties such as available bandwidth, delay, and loss, on the egress link with netem. We decided to place both, the QUIC client and server of the prototype implementation, on the same physical machine depicted on the left. So, timestamps can be taken from the same physical clock source.

We want to emulate random burst loss that is not caused by self-induced congestion, as experienced, e. g. on Internet paths with cross traffic. The stochastic process of netem's stateful loss models is advanced by packets. Therefore, we apply continuous background traffic to advance the loss process independently from the count of packets that are sent in the QUIC connection. For that purpose, we run the network test tool iperf3 alongside the modified quiche application. We use it as a packet generator sending UDP cross traffic with a rate of 10 Mbit/s continuously and bidirectionally through the same links as the QUIC connection.



(a) File size: 32 kB (b) File size: 1 MB

Fig. 4: Cumulative distribution of transfer completion times of non-incremental HTTP/3 transactions

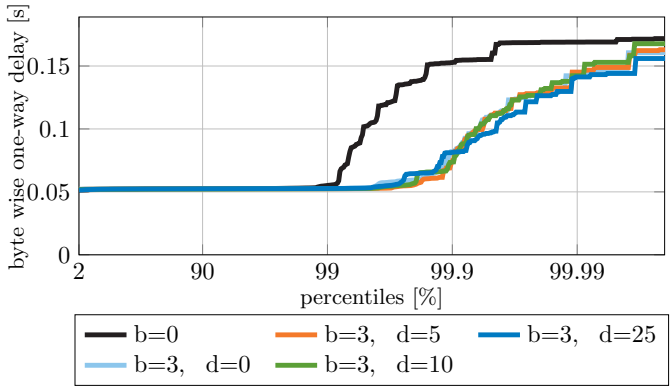


Fig. 5: HDR plot showing the statistical distribution of the measured byte-wise delay of a streaming web resource with a logarithmic horizontal axis

2) Network path parameters: All network path parameters are emulated bidirectionally with the same parameter set. We set the OWD to 50 ms and the path capacity to 10 Mbit/s. For the loss emulation, we use the Gilbert-Elliot loss model with the probabilities of 0.005 to enter the bad state, 0.25 to exit it, a probability to drop a packet of 0.95 in the bad state, and 0.005 in the good state respectively. The values are within the range studied in [12]. We limit the path parameter space to a single configuration, as Tetrys has already been studied with a wide range of path parameters [12], [16]. To make the congestion control less sensitive to loss, we use BBR2 as congestion control algorithm.

3) Non-incremental web assets: To assess the benefit of FEC for non-incremental web assets, we repeat file transfers 1000 times for each configuration and measure the TTC. We chose the file sizes 32 kB and 1 MB, corresponding to common workloads of web applications [27]. The measurement results are visualized in Figure 4. A general conclusion, which confirms previous assessments [12], is that FEC has a larger relative impact on smaller transactions. For those, the transfer time is within a similar range as the RTT. In our measurements, this effect can be seen when we compare Figure 4a and Figure 4b. Almost 40% of transfers without FEC ($b = 0$) need one retransmission, reflected by the knee at 0.2 s. A share of about 5% requires a second retransmission. With FEC protection, the number of retransmissions is halfed. The count of sent repair symbols only has a minor impact.

4) Incremental web assets: Previously, the byte-wise end-to-end OWD was identified as a key metric to assess the quality of incremental web transactions. Instead of transferring actual web ADUs, in the evaluation, we generate data in the server application with a custom binary format. It includes timestamps in the HTTP/3 response body. Thus, the HTTP/3 client application can measure the time a particular data segment took to arrive from the server application.

We mimic a streaming workload which continuously produces small ADUs. In the experiment, we generate 5 kB chunks at a rate of 60 Hz, and transfer 100 MB in total for each configuration. The results of the measurement campaign are plotted as an HDR plot in Figure 5. It shows that the delays for all studied configurations remain roughly at the path delay for 99.1% of the arriving bytes. Without FEC, values gradually increase. The theoretical time for a successful ARQ-based recovery is $\frac{13}{8}RTT$, which corresponds to about 160 ms. 0.1% of the data arrives after that. The delays of the data segments belonging to the percentiles in between partially are attributed to Fast Retransmit, i. e., when the receiving endpoint detects gaps in the packet numbers and sends duplicate ACKs, which leads to shortened loss detection.

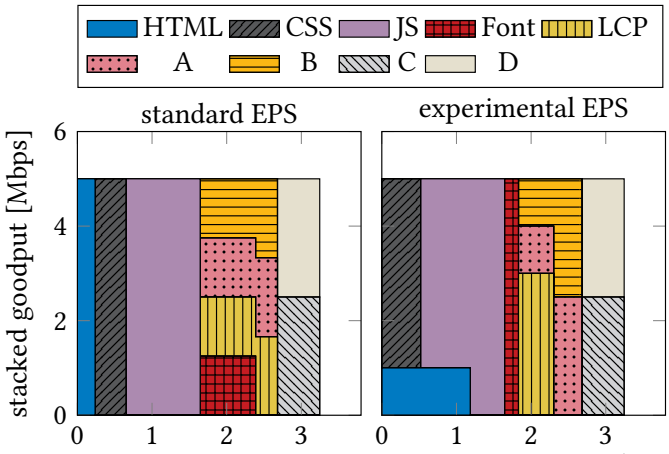
When BLT is set to 3, the delay remains at the 50 ms path delay for 99.8% of the transferred bytes. From there on, there is a gradual increase. For higher RDT settings, e.g., 25 ms, there is an initial raise in delay, exceeding that of the curves with smaller RDTs. A step at 75 ms is visible, corresponding to the additional waiting time caused by the delayed RSs. For the higher percentiles, higher RDT settings begin to outperform, likely attributed to the increased temporal separation from protected SSs.

5) Discussion: The results show a improvement in the studied loss scenario. For non-incremental workloads the TTC is improved, especially for smaller transfers, commonly found in web applications. Incremental workloads profit from FEC. Almost ten times as many stream segments arrive within no substantial delay, compared to the path's OWD. The RDT only has little influence on the delay. Albeit additional complexity is introduced by FEC, there are significant benefits. We anticipate that those are transferable to other relevant use cases, such as Media over QUIC or WebSockets [28], [29].

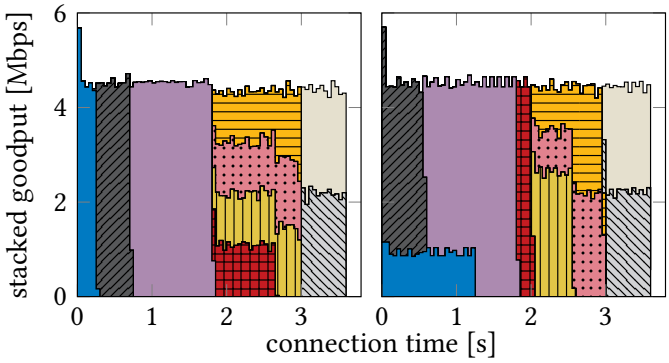
C. Hierarchical Link Sharing in QUIC

In this subsection, we compare our extended scheduling strategy against EPS' standard one. We model a sample web page and let a client retrieve it from a server on our testbed.

1) Example web site: Webpages may load hundreds of resources upon access: *The New York Times*, for instance, loads more than 150 [5]. With such a large number of parallel requests, illustrating the operation of our experimental EPS scheduler becomes more challenging. To ease understanding and show how finer scheduling control can be advantageous, we modeled a sample website consisting of an Hypertext Markup Language (HTML) document (148 kB), Cascading Style Sheets (CSS) (260 kB), JavaScript (JS) code (620 kB), font (116 kB), and five image files (177 kB each). One of



(a) Theoretical scheduling order at 5 Mbit/s goodput



(b) Goodput measured at the client with emulated path properties of 5 Mbit/s capacity, 10 ms RTT; the bin width is 50 ms

Fig. 6: Scheduling order using standard EPS parameters (Figure 1a) and experimental hierarchical extensions (Figure 1b)

these images represents the Largest Contentful Paint (LCP) element, whose ADU completion time we want to expedite for better end-user experience. The listed file sizes are at the 90th percentile for desktop web browsing in 2024 [27] only the JS resource has a median size.

2) Standard vs. experimental EPS scheduling: The priorities shown in Figure 1a are based on EPS signals that the Safari browser uses for each resource type and its behavior of always setting the incremental flag [4]. Despite that, the server does not leverage QUIC's stream multiplexing capabilities since the HTML, CSS, and JS files are on separate urgency levels. Only the remaining assets are sent in parallel and share the connection bandwidth equally. The expected scheduling order is illustrated on the left side of Figure 6a. It can be obtained from the consecutive application of algorithm III-B4 to determine $\mathcal{L}_s$. With the flat hierarchy as input and assuming all streams are active, $\mathcal{L}_s = \{\text{HTML}\}$ is returned as it has urgency 0. This remains so until it is fully transmitted and removed from the hierarchy, at which point the strategy begins returning the CSS file. More than one incremental stream is tied at the third urgency level, $\mathcal{L}_s$ becomes $\{\text{Font}, \text{LCP}, \text{A}, \text{B}\}$.

Usage of the `exp_w` and `exp_p` parameters, in contrast, enables resources to be conceptually grouped and assigned weights. In Figure 1b, we introduce parent classes for *Web assets* and *Images*, further making a distinction between elements *In viewport* and those that are not. Figure 6a shows the expected outcome on the right after applying the scheduling algorithm. Real measurements in Figure 6b closely match the intended outcome. Unlike before, we see that the HTML file is being loaded in parallel to *Web assets* at a 1 : 4 ratio. The CSS and JS files, in turn, are strictly loaded sequentially *within* their class. Per HMM fairness, the *Web assets* class is allocated the excess bandwidth once the HTML file completes. In a production environment, where the client is not making all requests simultaneously, the HMM principle would ensure the HTML file receives the full bandwidth until one of the other requests are made despite its low weight. Likewise, by giving the LCP image a larger weight, we ensure more of it arrives earlier than other images in the viewport.

3) Discussion: QUIC's RFC does not specify a general-purpose stream scheduler explicitly stating that priority signals ought to come from the application [1]. HTTP/3 requires QUIC, but it does not specify a prioritization mechanism either [30]. While EPS provides a minimal, tailor-made interface to control how streams are multiplexed, its usage in various browsers is still far from optimal [4]. Our proposed extensions to the EPS parameter set address the latent demand for weighted incremental streams and generalize HMM-fair scheduling for compatibility with EPS. This enables arbitrarily deep, classful stream traffic divisions where some of the nested classes follow strict priorities *while* other classes emit data incrementally. Although HTTP/2 dependency trees were deprecated due to their complexity and low adoption [25], we are confident that aggregating web resources by conceptual classes in lieu of dependencies is simpler. The performance of Core Web Vital metrics such as LCP is heavily influenced by how streams are multiplexed, so more levers to pull can aid in approaching optimal behavior. As the proof of concept implementation was done in QUIC, it is applicable beyond HTTP/3 applications.

VI. Related Work

Related work can be found in several active research areas. First, an overview of the state of integrating FEC in QUIC is provided. Next, we summarize research activity on HMM scheduling. Finally we list works on HTTP prioritization.

A. QUIC with Forward Error Correction

There exist RFC drafts and research articles discussing FEC extensions for QUIC.

1) Internet standard drafts: There are suggested extensions to the QUIC protocol that modify its packet header and divide its payload into chunks of SSs [31], [32]. Additionally, they introduce mechanisms to transmit RSs. Those methods do not allow frame-specific FEC mechanisms as they operate on the packet level. In contrast to those, the recent QUIRL paper and the related Request for Comments (RFC) draft do not modify the packet header [14]. Instead, they specify new transport parameters and additional frames. As a result, the protocol extension allows the use of the Tetrys coding scheme

which requires symbol acknowledgments to update the coding window of the encoder.

2) Research articles: A first study about the integration of FEC into QUIC focuses on the video streaming use case. The results show that Reed-Solomon and sliding window convolutional codes in particular are beneficial to decrease loss, improve OWD, and avoid rebuffering of video streams [8]. The paper rQUIC assesses bulk and website load times with QUIC extended with an XOR coding scheme. The main result is that especially website load times improve in case of lossy paths [9]. In QUIC-FEC, additional contributions are made. The authors measure the impact of QUIC with XOR, Reed-Solomon, and sliding window codes on file transfers of various sizes. They conclude that in particular short downloads profit from FEC in lossy communications in high-delay paths [10]. In FlEC, application-specific RS schedulers are suggested for bulk transfers, buffer-limited devices, and small messages. A Random Linear Code (RLC)-based coding scheme is used [11]. The recent work QUIRL evaluates benefits of FEC in QUIC for video streaming and file transfer workloads. The Tetrys coding scheme is used. The number of sent RSs relies on the detected median burst loss event size or a default value if the measurement is not yet available. A custom scheduler is introduced with the goal of protecting key video frames only, reflected as bigger SS bursts. The `MaxJitter` parameter expresses the delay between the burst and sending of the RSs. The RS scheduler for HTTP/3 is focused on non-incremental workloads and schedules RS when in application limited phases. The approach yields improvements of download completion times, especially under last flight losses, and video playback QoE. The evaluation of bulk transfer assesses the time-to-completion of transfers [12].

3) Comparison: The approach to embed FEC into QUIC presented in this work integrates many concepts from related works. The protocol extension and the coding scheme are similar to the one used by QUIRL [12]. Our work addresses incremental and non-incremental HTTP/3 workloads, while QUIRL is focused on the latter. The RS scheduling of non-incremental workloads is similar to the one presented in FlEC and QUIRL: RSs are sent in application limited phases [11], [12]. Due to the optimization for video streaming codecs, the RS scheduler from QUIRL is not universally applicable to other streaming applications. Still, our incremental workload scheduler is similar by sending RSs in application limited phases. Inspired by the `MaxJitter` parameter, we introduce the RDT, with the additional rationale of gaining statistical independence from correlated loss events showing a small positive effect in our evaluation. In QUIRL, the count of repair symbols is adapted with path statistics. Whereas in this work, we suggest extensions to the HTTP/3 priority signaling scheme to include RS scheduling parameters. They can be updated by `PRIORITY_UPDATE` frames, e. g. based on path characteristics, too. The FEC configuration is specific to streams, enabling heterogeneous workloads within a single connection. We assess the non-incremental case using typical transfer sizes found in web applications; QUIRL presents results for files of 50 kB and larger. In the QUIRL paper, improvements to the streaming performance are assessed using real applications and QoE metrics, whereas in this approach we present statistics of measured byte-wise OWD under synthetic workloads. For better reproducibility, we chose to use network emulation and a controlled testbed environment. In contrast, QUIRL partially uses Starlink for measurements. Only a subset of its source code is available. We intend to release the full code with the final version of this paper.

B. HMM Scheduling

Fair bandwidth allocation schemes similar to HMM find applications beyond HLS' qdisc. HLS was implemented in the INET framework for the OMNet++ discrete event simulator, yielding significantly less jitter when excess bandwidth is redistributed than in Linux's Hierarchical Token Bucket (HTB). In overloaded networks, the delay introduced by HTB was $\frac{2}{3}$ larger than HLS' [33]. Programmable hierarchical packet scheduling at the switch level was achieved on a single physical queue [34]. Similarly, for software-defined networks, an HMM-fair scheduler has been created [35]. HTB-like traffic shaping, on which HLS improves upon, appears promising for the slicing of 5G networks [36].

C. EPS-based HTTP Prioritization

Although established browsers use EPS signals disparately, they *do* adopt the scheme, even if coarsely and with room for optimizations [4]. Other protocols, too, such as HTTP Live Streaming, require this scheme. Investigations reveal its effectiveness for low-latency content delivery [37]. The weighted incremental scheduling is shown to improve LCP [37]. In another approach, different EPS-based schedulers that are dynamically chosen based on the network conditions reduce webpage load times [38]. Further ambitious approaches use reinforcement learning techniques [39].

VII. Conclusion

In this paper, we investigate ways to make QUIC faster. FEC is used to improve transmission delay caused by the recovery of lost packets and a hierarchical weighted round-robin scheduler enables finer control over the order of sent data. The mechanisms can be controlled by HTTP/3 applications with experimental EPS signals. The FEC mechanism relies on the Tetrys coding scheme, which is robust, in particular, to bursty loss patterns. For the redundant information, we provide suitable scheduling strategies and distinguish between non-incremental and incremental web workloads. The stream multiplexing algorithm, while remaining fallback-safe with existing EPS, supports weighted round-robin prioritization and introduces hierarchical scheduling classes. It is designed to be HMM fair on a byte-granular level.

We implemented the extended functionalities in the Cloudflare quiche library and assess their impact in a testbed environment. A benchmark shows that the FEC implementation is fast enough for many use cases. Using emulated network properties, we evaluate that significant timing improvements are

achievable for non-incremental and incremental web transfers. We defined an example website workload with realistic priority parameters and compare the download behavior between standard EPS and suggested experimental EPS. We found that the new scheduling approach improves stream parallelism, which makes HOL blocking less likely. The gained flexibility enables earlier completion of important web resources, reflected, for example, in an improved LCP metric.

In this work, concepts found in literature on FEC and stream scheduling are integrated into the web HTTP/3 stack and hereby achieved results demonstrate substantial improvements of transmission times. The source code is available online [13] to encourage additional research.

Acknowledgments

We thank Michael Hackl for his help.

This work was supported by the EU Horizon Europe programme, projects SLICES-PP (10107977) and GreenDIGIT (101131207), by the German Federal Ministry of Education and Research (BMBF), projects 6G-life (16KISK002) and 6G-ANNA (16KISK107), by the German Research Foundation, project HyperNIC (CA595/13-1), and by the Bavarian Ministry of Economic Affairs, Regional Development and Energy, project 6G Future Lab Bavaria.

References

[1] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000, 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9000

[2] F. Bulgarella *et al.*, "Performance measurements of QUIC communications," in *Proceedings of the 2019 Applied Networking Research Workshop*, 2019, pp. 8–14.

[3] A. Yu and T. A. Benson, "Dissecting performance of production QUIC," in *Proceedings of the Web Conference 2021*, 2021, pp. 1157–1168.

[4] J. Herbots *et al.*, "HTTP/3's Extensible Prioritization Scheme in the Wild," in *IRTF-ANRW*, 2024, pp. 1–7.

[5] C. Sander, I. Kunze, and K. Wehrle, "Analyzing the Influence of Resource Prioritization on HTTP/3 HOL Blocking and Performance." in *TMA*, 2022.

[6] R. Marx., T. De Decker., P. Quax., and W. Lamotte., "Of the Utmost Importance: Resource Prioritization in HTTP/3 over QUIC," in *WEBIST*. INSTICC, 2019, pp. 130–143.

[7] meta.ai, "HTTP Prioritization for Product Performance," 2024. [Online]. Available: http://bit.ly/3YjsMzK

[8] F. Michel, Q. D. Coninck, and O. Bonaventure, "Adding Forward Erasure Correction to QUIC," *CoRR*, vol. abs/1809.04822, 2018. [Online]. Available: http://arxiv.org/abs/1809.04822

[9] P. Garrido, I. Sanchez, S. Ferlin, R. Aguero, and O. Alay, "rQUIC: Integrating FEC with QUIC for robust wireless communications," in *GLOBECOM*. IEEE, 2019, pp. 1–7.

[10] F. Michel, Q. De Coninck, and O. Bonaventure, "QUIC-FEC: Bringing the benefits of Forward Erasure Correction to QUIC," in *2019 IFIP Networking Conference (IFIP Networking)*. IEEE, 2019, pp. 1–9.

[11] F. Michel, A. Cohen, D. Malak, Q. De Coninck, M. Médard, and O. Bonaventure, "FlEC: Enhancing QUIC with application-tailored reliability mechanisms," *IEEE/ACM Transactions on Networking*, 2022.

[12] F. Michel and O. Bonaventure, "QUIRL: Flexible QUIC Loss Recovery for Low Latency Applications," *IEEE/ACM Transactions on Networking*, 2024.

[13] "QUIC FEC EPS," 2025. [Online]. Available: https://github.com/holzingk/quic-fec-eps

[14] F. Michel and O. Bonaventure, "Forward Erasure Correction for QUIC loss recovery," 2023. [Online]. Available: https://datatracker.ietf.org/doc/draft-michel-quic-fec/01/

[15] B. Adamson *et al.*, "Taxonomy of Coding Techniques for Efficient Network Communications," RFC 8406, 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8406

[16] P. U. Tournoux *et al.*, "On-the-fly erasure coding for real-time video applications," *IEEE Transactions on Multimedia*, vol. 13, no. 4, 2011.

[17] J. Detchart, E. Lochin, J. Lacan, and V. Roca, "Tetrys: An On-the-Fly Network Coding Protocol," RFC 9407, 2023. [Online]. Available: https://www.rfc-editor.org/info/rfc9407

[18] S. M. Günther, N. Appel, and G. Carle, "Galois Field Arithmetics for Linear Network Coding using AVX512 Instruction Set Extensions," in *arXiv: cs.DC; cs.NI*, 2019.

[19] V. Roca, M. Watson, and A. C. Begen, "Forward Error Correction (FEC) Framework," RFC 6363, 2011. [Online]. Available: https://www.rfc-editor.org/info/rfc6363

[20] J. Iyengar and I. Swett, "QUIC Loss Detection and Congestion Control," RFC 9002, 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9002

[21] T. T. Thai, J. Lacan, and E. Lochin, "Joint on-the-fly network coding/video quality adaptation for real-time delivery," *Signal Processing: Image Communication*, vol. 29, no. 4, pp. 449–461, 2014.

[22] A. Ali *et al.*, "Bandwidth efficient adaptive forward error correction mechanism with feedback channel," *Journal of communications and networks*, vol. 16, no. 3, pp. 322–334, 2014.

[23] M. Belshe, R. Peon, and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)," RFC 7540, 2015. [Online]. Available: https://www.rfc-editor.org/info/rfc7540

[24] N. Luangsomboon and J. Liebeherr, "HLS: A Packet Scheduler for Hierarchical Fairness," in *2021 IEEE 29th International Conference on Network Protocols (ICNP)*. IEEE, pp. 1–11. [Online]. Available: https://ieeexplore.ieee.org/document/9651972/

[25] K. Oku and L. Pardue, "Extensible Prioritization Scheme for HTTP," RFC 9218, 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9218

[26] M. Nottingham and P.-H. Kamp, "Structured Field Values for HTTP," RFC 8941, 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc8941

[27] HTTP Archive, "2024 Web Almanac: HTTP Archive's annual state of the web report," 2024, accessed: 2025-03-04. [Online]. Available: https://almanac.httparchive.org/en/2024/

[28] L. Curley *et al.*, "Media over QUIC Transport," Internet Engineering Task Force, Internet-Draft draft-ietf-moq-transport-10, 2025. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-moq-transport/10/

[29] R. Hamilton, "Bootstrapping WebSockets with HTTP/3," RFC 9220, 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9220

[30] M. Bishop, "HTTP/3," RFC 9114, 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9114

[31] V. Roca *et al.*, "Sliding Window Random Linear Code (RLC) Forward Erasure Correction (FEC) Schemes for QUIC," Informational Draft, 2020. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-roca-nwcrg-rlc-fec-scheme-for-quic-03

[32] D. Moskvitin *et al.*, "Forward Erasure Correction for Short-Message Delay-Sensitive QUIC Connections," Standard Draft, 2023. [Online]. Available: https://www.ietf.org/archive/id/draft-dmoskvitin-quic-short-message-fec-00.html

[33] A. Iyidogan, M. Bosk, and F. Rezabek, "Hierarchical Resource Sharing and Queuing in OMNeT++ and INET Framework," in *OMNeT++ Community Summit*, 2022.

[34] Z. Zhang *et al.*, "vpifo: Virtualized packet scheduler for programmable hierarchical scheduling in high-speed networks," ser. ACM SIGCOMM '24, 2024. [Online]. Available: https://doi.org/10.1145/3651890.3672270

[35] D. Ran *et al.*, "HSDBA: a hierarchical and scalable dynamic bandwidth allocation for programmable data planes," *Frontiers of Information Technology & Electronic Engineering*, vol. 25, no. 10, 2024.

[36] P. Raussi *et al.*, "Prioritizing protection communication in a 5G slice: Evaluating HTB traffic shaping and UL bitrate adaptation for enhanced reliability," *The Journal of Engineering*, vol. 2023, no. 9, 2023.

[37] A. Gupta *et al.*, "Improving HTTP/3 Quality of Experience with Incremental EPS," 2024. [Online]. Available: https://arxiv.org/abs/2403.04074

[38] Y. Sasaki and K. Sato, "Reducing Web Page Load Time by Using a Dynamic Scheduling Method Based on Network Environment," *IEICE Communications Express*, vol. 13, no. 4, pp. 122–125, 2024.

[39] K. Wong and L. Cui, "Fine-grained HTTP/3 prioritization via reinforcement learning," *Computer Networks*, vol. 233, p. 109880, 2023.