# Resource Multiplexing and Prioritization in HTTP/2 over TCP versus HTTP/3 over QUIC

Robin Marx[1,2], Tom De Decker[1], Peter Quax[1,3], and Wim Lamotte[1]

[1] Hasselt University – tUL – EDM, Diepenbeek, Belgium
{first}.{lastname}@edm.uhasselt.be
[2] SB PhD fellow at FWO, Research Foundation Flanders, #1S02717N
[3] Flanders make

**Abstract.** Modern versions of the HTTP protocol, such as HTTP/2 over TCP and the upcoming HTTP/3 over QUIC, use just a single underlying connection to transfer multiple resources during a web page load. The resources are divided into chunks, optionally multiplexed on the connection, and reassembled at the receiver's side. This poses challenges, as there are many different ways simultaneously requested resources can share the available bandwidth, and not all approaches perform equally well with regards to achieving low loading times. Making matters worse, HTTP/2's prioritization system for directing this multiplexing behaviour is difficult to use and does not easily transfer to the new HTTP/3. In this work, we discuss these challenges in detail and empirically evaluate the multiplexing behaviours of 10 different QUIC implementations, as well as 11 different prioritization schemes for HTTP/3. We find that there are large differences between strategies that can have a heavy impact on page load performance, of up to 5x load time speedup in specific conditions. However, these improvements are highly context-sensitive, depending on web page composition and network conditions, turning the best performers for one setup into the worst for others. As such, we also critically evaluate the ability of the newly proposed HTTP/3 prioritization mechanism to flexibly deal with changing conditions.

**Keywords:** Web Performance, Resource Prioritization, Bandwidth Distribution, Network Scheduling, Measurements

## 1 INTRODUCTION

The HTTP protocol has undergone some highly impactful changes in the past few years, starting with the standardization of HTTP/2 (H2) in 2015 and now the upcoming finalization of HTTP/3 (H3), barely five years later. This rapid evolution is driven mainly by the need for improvement in two key areas: performance and security, and this work focuses on the former. When loading web pages over HTTP, browsers typically request a large amount of different resources. These resources are spread over a range of different types, including HTML, CSS, JavaScript and image files. Over HTTP/1.1, only one of those resources can be in-flight on the underlying TCP connection at a time, holding up all further resources behind it until it is fully downloaded. This is called the Head-Of-Line (HOL) blocking problem. As a workaround to achieve better web page loading performance, browsers open up to six parallel HTTP/1.1 connections per domain, each carrying one resource at a time. By distributing resources over multiple domains this number of connections can grow to 30 or more. While this typically gives the intended benefit of faster page loads, it is also inefficient, as each TCP connection requires some memory and processing. Additionally, it introduces bandwidth contention problems, with each individual connection vying for their share. As TCP's congestion control mechanisms work on a per-connection basis and often use packet loss as their main backoff signal, this massive parallelism can lead to increased packet loss rates and fairness issues with other applications using less TCP connections.

As such, one of the main goals of H2 was to allow the multiplexing of a web page's resources on a single underlying TCP connection [11]. To this end, H2 subdivides resource payloads into smaller chunks which are prefixed with their unique resource identifier, allowing data from different resources to be interleaved on the wire. While this resolves HTTP/1.1's HOL blocking problem, it introduces new challenges. The question now becomes how exactly the individual resources should be multiplexed and scheduled on the single connection. It turns out that this depends on the specific resource: some file types (such as JavaScript or CSS) typically need to be downloaded in full by the browser before they can be used and executed. As such, it makes sense to send them sequentially, not multiplexed with data from other resources. On the opposite end, resource types such as HTML and various image formats can be processed and rendered incrementally, making them ideal targets for heavy interleaving with data from other resources. Getting these resource priorities right is key to achieving good web page loading performance [19,9]. To allow browsers maximum flexibility in this area, H2 includes a complex prioritization mechanism, using a so-called "dependency tree", to steer which resources should be sent first and how.

In practice, this dependency tree mechanism has turned out to be overly complex and difficult to use, for browsers and servers alike [8]. Few implementations use its full potential and several employ sub-optimal tree layouts, leading to higher page load times. Additionally, switching to a single underlying TCP connection surfaces the fact that TCP has a Head-Of-Line blocking problem of its own when faced with packet loss. As TCP is a reliable and strictly ordered protocol, even a single packet loss can block all other packets behind it, waiting for its retransmission. This is inefficient for H2, as at that layer the blocked packets can contain data for other H2 resources, and as such do not necessarily have to wait for the lost packet's retransmission to be useful to the browser.

Solving TCP's HOL blocking problem is one of the main reasons for the new QUIC transport protocol [4]. While TCP regards all its payload data as a single, contiguous and opaque byte stream, QUIC is instead aware of multiple, fully independent byte streams (and thus HTTP resources) being transported over a single connection. As such, QUIC can provide features such as reliability and in-order data delivery on a per-stream basis; it is no longer tied to the global connection level like TCP. While this ostensibly solves TCP's transport level HOL blocking problem, it is unclear how much of an advantage this delivers in practice and if its benefits hold in all conditions. Another consequence of QUIC's fundamental departure from TCP's single-stream model, is that it becomes difficult to impossible to port some application layer protocols that currently rely on TCP behaviour to the new transport protocol. A key example of this fact is H2, which does not have a straightforward mapping onto QUIC. The discrepancies are so large in fact, that a new version of the protocol, termed H3, is being developed in tandem with QUIC. While H3 retains most of the semantics and high level features of H2, the underlying systems for aspects such as HTTP header compression, server push and the aforementioned resource prioritization have been substantially reworked to deal with QUIC's specifics. As such, the introduction of these new protocols leads to many unanswered questions with regards to performance and best practices.

In this text we continue the groundwork from our previous publications on HTTP/2 and HTTP/3 prioritization [6,19] and contribute new evaluation results for the QUIC protocol. While QUIC and HTTP/3 are conceptually bound together, they are in essence still separate protocols. As such, we will first discuss them individually and then combine our findings in the discussion. Starting at the transport layer in Section 2, we look at how ten different QUIC implementations implement multiplexing and data retransmissions on the independent byte streams. We find that there are large differences in how various stacks employ the new protocol's options. In Section 3 we provide new data concerning QUIC's HOL blocking behaviour. We find that QUIC provides definite benefits over TCP in this area, but that they are highly dependent on the underlying resource multiplexing behaviour. We then explore our application layer results from previous work more deeply in Sections 4 and 5, discussing the problems with H2's prioritization setup and why it had to change for H3. We present the results of evaluating 11 H3 prioritization schemes on 42 web pages, to again show the optimal approach to be highly context dependent. We end by combining insights from both QUIC and HTTP/3 and discuss how the results from our previous work helped initiate a radical change in H3's planned approach to prioritization. All our source code, results and visualizations are made publicly available at https://h3.edm.uhasselt.be.

## 2 QUIC Resource Multiplexing

### 2.1 Background: Resource Streams

QUIC supports the concept of multiple, independent byte streams being active on a connection at the same time. These streams are a rather abstract concept in QUIC itself, but map nicely to for example individual resource requests and responses at the HTTP layer. This is in stark contrast with TCP, which sees all the data it transports as part of a single opaque byte stream, even if it is in fact carrying data from multiple files for protocols such as H2.

This can be easily understood by looking at the protocols' mechanisms under the hood. TCP for instance, uses a *sequence number* in its packet header, indicating the starting byte offset of the data in the current packet within the larger stream. For example, a TCP packet 1400 bytes in size with a sequence number of 10, carries byte 10 up to and including byte 1409 of the payload. QUIC takes a different approach, not including the payload metadata in its packet header, but rather using an additional binary framing layer. QUIC's STREAM frames contain a stream identifier (ID), to indicate which byte stream this frame belongs to, and both offset and length fields. However, these offsets are separate per stream: if we have a STREAM frame for stream A with offset 10 and length 1400, we can also have a subsequent STREAM frame for stream B with offset 10 and length 1400 and they would carry completely independent data. In TCP, stream B's data would then rather have been given offset 1410, as it was being sent directly after A's data.

While it adds flexibility, QUIC's setup also provides new challenges. Where TCP just needs to transport its single byte stream as it receives it from the application layer, QUIC now has to decide which of the multiple in-progress streams gets to send data in each outgoing QUIC packet. Put differently: it has to decide on a multiplexing and scheduling approach. As we will see in Section 4, this decision can be driven by the application layer, for example by H2's and H3's prioritization system. However, since QUIC is a standalone transport protocol, intended for use besides HTTP as well, it is useful to consider these mechanics purely on the transport layer as well. It is important to note that while QUIC includes the abstract concept of streams, it knows not of the semantics tied to individual streams and thus also cannot derive relative stream priorities itself. Put differently, QUIC treats all streams as equally important.

Consider then some possible approaches for two streams, A and B. One possibility is a *sequential* scheme, in which we would keep sending stream A's (available) data in full, before starting to transport B. Another option is a *Round-Robin* (RR) scheduler, in which we send a limited amount of data for A before switching to B for some time, moving back to A afterwards and so on. There are many possible RR variants, depending on how much data each stream may send before switching. For example, the scheduler could switch after 40 packets, after just one packet, or could even aggregate a smaller STREAM frame from both A and B into a single QUIC packet. The question is then: which of these schemes and their variants performs best in which circumstances?

This question becomes even more complex when we consider reliability. Just like TCP, QUIC utilizes retransmissions based on acknowledgements and timeouts to ensure reliable data transfers. Unlike TCP, these retransmission are now also done on a per-stream basis. While QUIC still acknowledges full packets, endpoints are expected to keep track of which STREAM data was in which packet. As such, QUIC does not necessarily need to retransmit full packets, but only individual STREAM frames. In fact, not even that is needed: QUIC implementations only need to keep track of the ranges of a byte stream (offset + length) that have not been acknowledged. When packet loss is detected, the lost ranges can be re-packaged into new STREAM frames at will (e.g., two smaller ranges that were previously sent in two STREAM frames across two packets can just as well be combined into one large range in one packet for retransmission). As such, QUIC implementations also need to make decisions in how to schedule the retransmission of this lost data in comparison to the “normal” data. We will consider three main **Retransmission Approaches** (RAs), which are the ones observed in our experiments in Section 2.3. In the following example sequences, we assume a default RR scheduler over streams A, B, C and D, where streams A and B suffer some losses:

– **RA #1: Default scheduling**: to be retransmitted data is simply seen as ”normal” data and is sent when the stream is next selected by the scheduler as part of normal operating procedure. It does not receive special treatment. An example sequence would be ABCDABCD.
– **RA #2: Priority retransmissions with default scheduling**: streams with lost data are given a higher priority, and between them the default scheduling applies. In the example with the default RR scheduler, it will first perform RR between streams with losses before going back to RR across all streams. An example sequence would be ABABCDCD.
– **RA #3: Priority retransmissions with special scheduling**: streams with lost data are given a higher priority, and between them a special scheduling applies. In the example with the default RR scheduler, this approach can first utilize a sequential (or other) scheduler between streams with losses, before going back to RR across all streams. An example sequence would be AABBCDCD. Conversely, in the case of a default sequential scheduler, RR could be used for lossy streams, resulting in for example ABABCCDD.

It should be evident that there is no immediate clear winner among these schedulers and RAs: each has their own tradeoffs and will perform better or worse depending on use case and on the specific types and frequencies of loss (e.g., bursty vs single packets). Given this large parameter space, we decided to evaluate the default choices made by current QUIC implementations. These will dictate the off-the-shelf behaviour and performance seen from QUIC stacks when not utilizing external prioritization signalling.

## 2.2 Experimental Setup

At the time of writing (January 2020), there are approximately 20 publicly announced IETF QUIC implementations[4], many of them tied to large internet companies. Note that we only look at the IETF QUIC proposed standard, not Google’s original version of the QUIC protocol, which is implemented by much less different parties. Note too that, to our knowledge, we are the first to perform this type of evaluation on multiple IETF QUIC implementations, related work being limited to either Google’s original QUIC implementation or a single IETF QUIC stack.

[4] https://github.com/quicwg/base-drafts/wiki/Implementation

Most of the IETF QUIC implementations provide a publicly hosted endpoint for testing, or are open source. After filtering out obviously broken, outdated, and closed source implementations without a public endpoint, we are left with 10 implementations: *aioquic*, *quiche* (Google), *lsquic* (LiteSpeed), *msquic* (Microsoft), *mvfst* (Facebook), *ngtcp2*, *picoquic*, *quiche* (Cloudflare), *ats* (Apache Foundation) and *quicly* (Fastly). Note that there are two implementations named *quiche*. Since Cloudflare named theirs first, we will henceforth refer to Google's quiche as simply *google*.

To execute our tests, we choose the *aioquic* client, as it implements all of QUIC's features, has excellent interoperability with the other implementations and is written in Python, which allows us to easily tweak its behaviour as needed. We run this client in a debian docker container and point it at the public endpoints to simultaneously request one to ten files, of various sizes between 1KB and 10MB. While we use H3 as a means to request these files out of convenience, no prioritization information is passed from H3 to the QUIC layer, allowing us to assess the transport layer's default behaviour.

We execute all our tests on two WAN networks. Firstly, on the Hasselt University network, providing 1 Gbps downlink and 10 Mbps uplink capacity. Secondly, on a residential Wi-Fi network, providing 35 Mbps downlink and 2 Mbps uplink. The first network allows us mainly to see multiplexing behaviour in optimal situations, while the second provides more insight in retransmission behaviour, as packet loss rates are somewhat higher on that network. All test permutations are run at least 10 times. For our analysis, we employ the logging output of the *aioquic* client in the *qlog* format [12]. *qlog* is a structured, JSON-based format which includes highly detailed endpoint event logs. We process the *qlog* events with custom scripts and visualize them using the qvis toolsuite at https://qvis.edm.uhasselt.be. For this, we contribute a new visualization to qvis, termed the "multiplexing graph", which helped produce the following images in this work.

### 2.3 Multiplexing Results and Discussion

Our main results are summarized in Figure 1. We show the singular experiment configuration of ten simultaneously requested and downloaded 1MB files across all tested endpoints. We visualize a single representative trace per endpoint. For *picoquic*, we include results before and after a major server-side code change (*picoquic_alt*). For *mvfst*, we include results after changing client-side parameters (*mvfst_alt*). Endpoints are loosely grouped by similar behaviour.

Figure 1 plots received QUIC STREAM frames (colored per individual stream) horizontally appended, thus hiding any inter-packet arrival time gaps. While this hides some contextual information, it helps to better view the overall multiplexing behaviour. Large contiguous blocks of the same color indicate a more sequential transfer for that stream, while rapid changes in color mean that an RR scheduler variant is being used. Black blocks are shown beneath STREAM frames that contain retransmitted data. These are identified by continuously tracking gaps in each stream's byte ranges (assuming a sender always sends data in-order per stream). Whenever a new frame fills a gap that was created sufficiently long ago, it is considered a retransmit. If it fills a gap that was created shortly before, it is assumed to be a re-ordered frame caused by network jitter. As such, areas without black blocks beneath show the endpoint's default multiplexing behaviour, while the rest allows us to assess their retransmission approaches (RAs). Note that the traces shown in Figure 1 are representative for each endpoint and that the trends discussed below were consistent across all traces unless otherwise mentioned.

We can observe some general trends in Figure 1. Firstly, a majority of implementations use a RR scheduler, with only two stacks opting for a sequential approach. Within the RR group, most use the fine-grained option of switching streams on each individual packet. Only *msquic* and *google* choose larger contiguous blocks of a fixed size (respectively 4 and 14 packets). *lsquic*'s

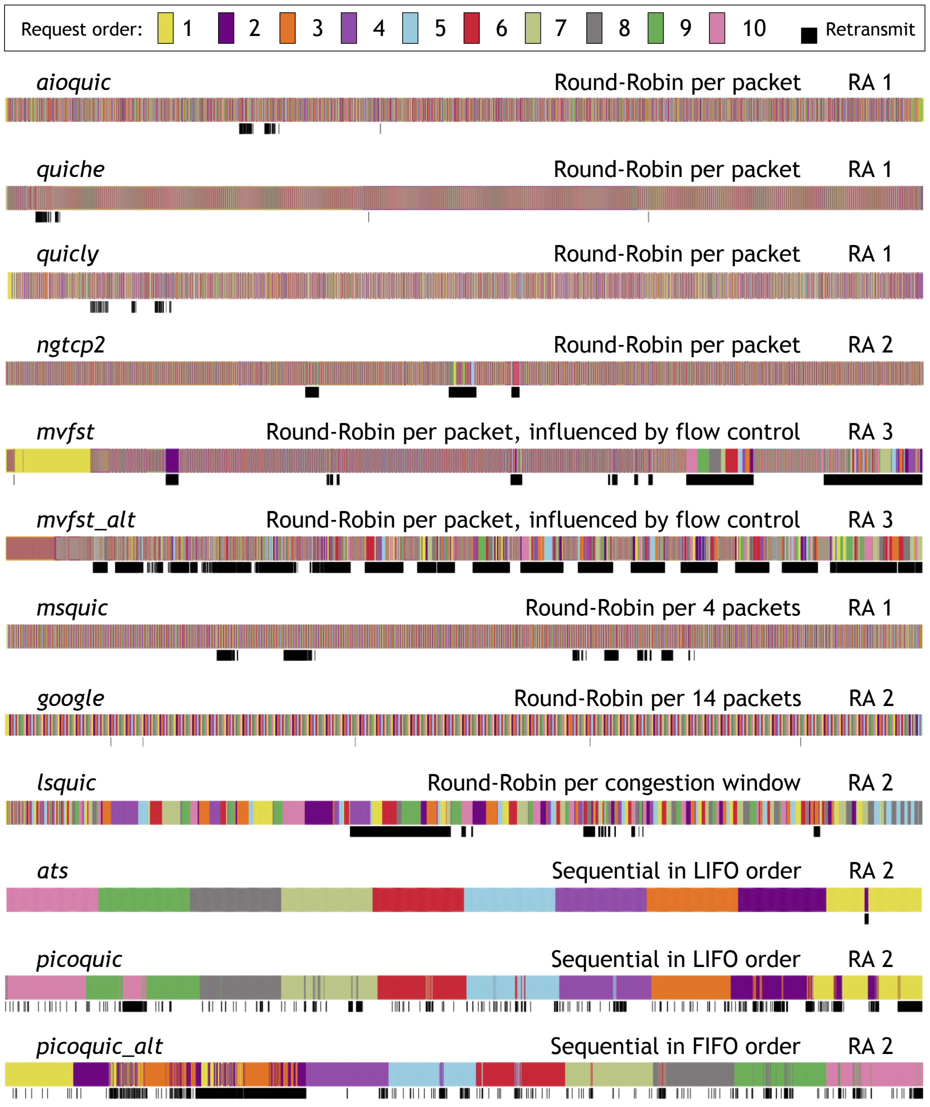

Fig. 1: QUIC endpoint multiplexing behaviour for ten simultaneously requested resources of 1MB each. Per endpoint, the top line indicates the multiplexing of individual resource data (each small rectangle is one STREAM frame). Black areas in the bottom line indicate which frames in the top line contain retransmitted data. STREAM frames arrive at the client from left to right.

behaviour seems strange at first, but it was confirmed by its implementer that it is intentional: for ease of implementation and increased server performance, it chooses its block size based on the current congestion window size. Whenever data can be sent, the next stream is selected, which sends all its available data filling the congestion window. This explains why the trace starts with fine-grained RR and gets coarser as time progresses, only to condense again when loss is detected.

Secondly, the RAs are about evenly split between #1 and #2. The key visual difference between the two, is that for #1, the black areas are usually interleaved with small white gaps, where data from non-lossy streams is being sent, indicating that the retransmissions don't receive absolute priority. This is for example visible in the *quicly* trace. Conversely, for #2, the black areas are mostly contiguous, indicating the implementations give highest priority to retransmitted data. A good example is the first big black block in the *lsquic* trace. Similar behaviour can be seen on the right side for *ats* and *picoquic*: retransmittable purple data immediately interrupts the current yellow stream.. Finally, there is only a single implementation, *mvfst*, that opts to change its behaviour when retransmitting data as an example of RA #3. Where it normally employs per packet RR, it switches to a fully sequential approach when there was lost data. We will discuss the wider impact of these general findings in Sections 3 and 6.

Next to the global findings, there are also several per-implementation quirks to be discovered. Firstly, both *ats* and *picoquic* employ a sequential scheme, but process the incoming requests in a Last-In First-Out (LIFO) order. Put differently, the tenth requested resource (pink) is sent first. While this might make sense in some use cases, it is generally not viewed as an optimal strategy. After conferring with the implementers, *ats* indicated they were aware of the behaviour, but are waiting for H3's new prioritization scheme to change their approach. On the other hand, *picoquic* was not expected to exhibit this behaviour. The maintainer implemented a fix[5], instead enforcing a First-In First-Out (FIFO) approach, shown in *picoquic_alt*. Similarly, *ngtcp2* might at first glance look like it uses RA #3, as it seems to switch to a sequential mode for the later retransmits. After conferring with the author, this turned out to be due to a bug in their fair queuing implementation, which was subsequently fixed[6]. Now, all *ngtcp2*'s retransmissions use per-packet RR as intended, making it an example of RA #2 (trace of the updated implementation not shown due to size limits).

Secondly, *mvfst* consistently showed a large sequential transfer for the first requested stream (yellow) at the start of all its traces, before switching to RR. With help from the implementers, we were able to track this down to QUIC's flow control mechanisms. Like TCP, QUIC includes dynamic flow control limits to prevent a fast sender from overflowing buffers at the receiver. However, where TCP only has a single, connection-wide flow control window (sometimes called the receive window), QUIC instead has multiple separate flow control limits: one for the connection as a whole like TCP, but also one for each individual stream. This feature lead to a complex sequence of interlocking behaviours which we will now trace step by step:

1. In our test setup, the *aioquic* client sets the connection-level flow control limit to 1048576 bytes (1MiB), but also sets the stream-level flow control limits for each individual stream to the same amount. This is done during the connection's handshake.
2. Our client sends ten simultaneous requests on ten streams, each for the same file of exactly 1000000 bytes (1MB) in size.
3. The *mvfst* server processes each request individually as it comes in and works with a complex internal buffering system that fills the buffers up to the current flow control limits.
4. The server processes the request for the first file and puts it into the buffer in its entirety, since the stream-level flow control limit for the first stream (1048576 bytes) is larger than the requested file's size (1000000 bytes). This leaves just 48576 of headroom with regards to the over-arching connection-level flow control.
5. The server processes the request for the second file and puts 48576 bytes of its data into the buffer, reaching the connection-level flow control limit (1048576 bytes).

[5] https://github.com/private-octopus/picoquic/commit/a66a5d0a0b02416f9fde46bb0c447bcc0b7abd60
[6] https://github.com/ngtcp2/ngtcp2/commit/4f705093d4c8f1ec5b231c2a1b557a6c966bc2f3

6. The server now incrementally processes the requests for the other eight files, but does not buffer their data, because the connection-level flow control limit is already reached.
7. The server starts sending, multiplexing with an RR scheduler between the data from just the first two streams, as can be seen at the very start of the *mvfst* trace. As the data from the second stream in the buffer soon runs out however, we end up just sending data from stream one.
8. After a while, the server receives an update to the connection-level flow control allowance from the client. At this point, it re-fills its internal buffers. As it now does have knowledge of all ten streams, it starts properly utilizing the RR scheduler across all of them.

We were able to easily confirm that this was the problem by changing the initial per-stream flow control limits to $\frac{1}{4}$th of the connection-level flow control. The result can be seen in the *mvfst_alt* trace, which shows the server multiplexing data for the first four streams from the start of the connection. Note that, while the client's initial flow control setup was the same for all endpoints, only *mvfst* exhibited this behaviour. Its implementers indicated that, for a production deployment, it would make sense to limit the amount of data that can be buffered per stream (for example to 64KB) if the intent is to use a RR scheduler. Further testing from our part on public *mvfst*-backed endpoints however, such as facebook.com and fbcdn.net, showed that they did not yet include this configuration change. It is interesting to note that flow control can be used by the client as a coarse prioritization/scheduling mechanism to dictate server behaviour, even overriding the server's normal intentions. For example, a RR server can be forced into sequential behaviour if the client only gives one stream flow control allowance at a time.

Thirdly, we looked at how implementations distribute STREAM frames over packets. While most stacks produce full-sized packets, all containing just a single STREAM frame for one stream, some implementations showed other behaviour. Notably *google* and *mvfst* had bugs that caused them to produce multiple, smaller STREAM frames for the same stream per packet. For example, one QUIC packet could contain 4 STREAM frames for stream one, sized 7, 500, 9, and 700 bytes respectively. The *mvfst* bug was since fixed[7], with Google indicating theirs was due to faulty re-use of HTTP/2 framing logic, which would be resolved in the future. Relatedly, *mvfst* was the only implementation we observed that occasionally multiplexed STREAM frames from **different** streams into a single QUIC packet. This only happened during retransmits, when for example stream one did not have enough retransmittable data to fill a full packet, which was completed with data from the next stream. Strangely, we also observed *mvfst* occasionally sending very small packets (e.g., 40 bytes). The reason was that *mvfst* adheres strictly to the current congestion window, which it measures in bytes instead of full packets. In some cases, the congestion window was not large enough to support another full packet, leading to a single smaller packet being sent instead. This sparked an interesting discussion with other implementers, who all indicated they instead rounded up to the next full packet in these situations. They felt that this (slight) overshooting of the congestion window was a good tradeoff between implementation complexity, correctness and performance. Note that, while it may seem that *mvfst* contained the most unexpected behaviours, this is mainly because it is a highly advanced implementation with complex logic. Additionally, its implementers were very receptive to our feedback and continued discussions with the authors of this work, leading to more in-depth scrutiny from our side over time. In practice, next to *quiche*, *mvfst* is the only QUIC implementation that has seen wide deployment and use in production with the facebook mobile app[8].

Finally, we feel it is important to note that all tested implementations are active works in progress and that these results are not necessarily representative of the final products these

[7] https://github.com/facebookincubator/mvfst/commit/ec9d1ccd2088c64319f743541b32789bb18ae2dc
[8] https://www.youtube.com/watch?v=8lYHNzoPS2o

companies may offer or deploy. The benefits however, of testing these implementations early, are manifold. Firstly, it helps to identify subtle bugs and quirks that can nevertheless have a noticeable impact. Secondly, it demonstrates differences between implementations, allowing implementers to re-assess their approaches. Thirdly, it highlights the inherent complexity of the QUIC protocol and the usefulness of specialized tooling and visualizations to perform debugging and analysis. Fourthly, and most importantly, it poses the question if the QUIC protocol should define a standardized interface to allow the application layer to manipulate QUIC's multiplexing behaviour. At the moment, the proposed QUIC specification [4] indicates that implementations **should** provide such an interface, but does not specify what such an interface should look like. As such, we end up not only with implementations choosing different default approaches, but also implementing different APIs (if they provide an API at all), making it more difficult for application layer protocol implementations to swap underlying QUIC stacks. Given that QUIC software is expected to be much more heterogeneous in behaviour than TCP deployments, this feels like an important point of action, allowing easier re-use of for example HTTP/3 implementations. For example, the recent "Pluginized QUIC" paper [3] can provide a possible starting point for a flexible API integration for this purpose. Further discussion on which multiplexing approach is optimal for performance follows in Section 5.3 and in the following exploration of Head-Of-Line Blocking.

## 3 QUIC Head-Of-Line Blocking

### 3.1 Background: Intra-Stream Blocking

Both QUIC and TCP are reliable protocols that deliver their received transported data to an application layer preserving the exact order in which that data was passed to the sender for transport. As seen in Section 2.1, TCP enforces this ordering across the entire connection as it collapses all data into a single byte stream, while QUIC instead only strictly orders data per individual byte stream utilizing separate offsets in STREAM frames. Put differently, in TCP, a packet A sent before B, is always passed to the application layer before B, while in QUIC the order may well become B and then A, if the packets contain data for different streams and A becomes delayed. It follows that packet loss can thus create gaps in the byte streams, which can only be filled by retransmitting the lost data. In these instances, it is possible that data succeeding the gap is correctly received, for example if the gap is caused by an accidental loss of just one packet. However, the correctly received data cannot yet be passed on to the application layer, as this would break the ordering requirement. As such, even a single lost packet can conceivably block other packets behind it. This can be sub optimal for performance if the blocked packets contain data that could be processed independently from the lost data. This is called the Head-Of-Line blocking problem and it is illustrated in Figure 2.

Because QUIC is no longer tied to a strict, connection-level ordering of its data, it is often said that it solves TCP's Head-Of-Line blocking problem. However, while it is true that QUIC removes TCP's inter-stream (connection-level) HOL blocking, it is still vulnerable to intra-stream HOL blocking. Put differently, if at the receiver QUIC stream A currently has a gap in its byte stream from offset 5000 to 6300, but has received and buffered stream A bytes 6301 to 40000 correctly, it still needs to wait for the gap to be filled. As soon as that happens, it can pass on bytes 5000 to 40000 in one large transfer to the application layer for processing, but not before. In this case, we could say 33699 bytes had been HOL blocked on this QUIC stream. The key advantage of QUIC over TCP is that if there were another QUIC stream B in progress, it would not be HOL blocked by loss on stream A. Note the exception to this: if QUIC implementations would multiplex STREAM frames of multiple streams into a single packet and that packet

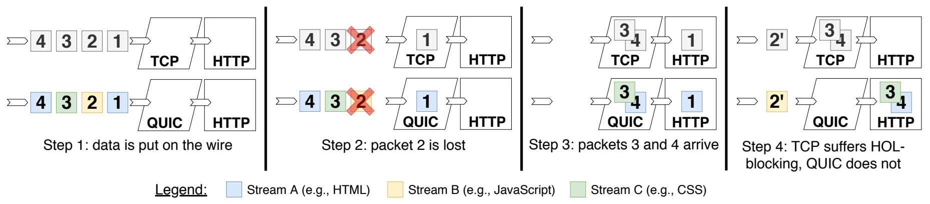
TCP
HTTP
QUIC
Step 1: data is put on the wire
Step 2: packet 2 is lost
Step 3: packets 3 and 4 arrive
Step 4: TCP suffers HOL-blocking, QUIC does not
Legend:
Stream A (e.g., HTML)
Stream B (e.g., JavaScript)
Stream C (e.g., CSS)

Fig. 2: Head-Of-Line blocking in TCP vs QUIC. Lacking knowledge of the three independent streams, TCP is forced to wait for the retransmit of packet 2 (2'). QUIC can instead pass packets 3 and 4 to HTTP immediately, where they are processed before packet 2' (source: [6])

is lost, this does lead to (limited) inter-stream HOL blocking. As such, it is considered bad practice to multiplex data from multiple streams into a single QUIC packet. Absent this caveat however, on QUIC, the application layer should have to wait less time between receiving bursts of processable data from the transport layer than it would for TCP under the same network conditions. In the case of HTTP, this could then potentially translate into for example the browser being able to show a picture to the user sooner over QUIC.

## 3.2 Experimental Setup

Overall, it is difficult to assess whether QUIC's removal of inter-stream HOL blocking always has the intended effects and how much it actually reduces observed HOL-blocking in practice, due to the continued presence of intra-stream HOL blocking. Additionally, we can already intuitively predict that the chosen multiplexing behaviour will also influence the occurrence of this intra-stream HOL blocking. Consider that, with a purely sequential approach, there is always only a single stream in progress at a time and this stream will thus always HOL block itself upon packet loss. As there are no other streams active at the same time, there is no other data to provide to the application layer while waiting for the single stream to become unblocked. With an RR scheduler this is different, as simultaneously active streams that are not HOL blocked themselves can continue to make progress from the application layer's perspective. As such, we can predict that with an RR approach, the application layer should more frequently get smaller batches of data to process incrementally, versus rarer but larger bursts for a sequential scheduler.

To assess the actual impact of QUIC's inter-stream HOL blocking removal, we would ideally need to run the same experiments over both TCP and QUIC over identical network setups. However, this is difficult in practice, since network simulations are rarely deterministic and H3 does not support TCP (nor does H2 QUIC). Instead, we devise a different approach by employing the rich output logs from the previous experiment described in Section 2.2. The *qlog* format [12] utilizes the `data_moved` event to indicate when a given range of stream data was actually passed from the transport to the application layer. Additionally, the logs contain `packet_received` events which specify when STREAM frames were received. Correlating these two event types gives us a good indication of intra-stream HOL blocking at the QUIC layer (e.g., a `packet_received` not immediately followed by a `data_moved`, indicates data was blocked).

Subsequently, we can also use these events to simulate what would happen if QUIC would suffer from the same inter-stream HOL blocking as TCP. At the first encountered gap in one of the streams, we start pretending `data_moved` events from other streams have not proceeded and their data is blocked behind this stream's gap, as would be the case in TCP. We do this until this first gap was filled through retransmits, and then cascade the held-back `data_moved` events for other streams until the next tracked gap. As new gaps could have occurred while waiting behind the first gap, we track gaps in a contiguous manner, keeping a rolling log of missing stream data and which `data_moved` events they are blocking within TCP semantics. This approach allows us to derive a good estimate of TCP HOL blocking behaviour based on the QUIC traces, meaning we can now compare them on identical network setups and experiment parameters. It also allows us to confirm our intuition that the scheduling mechanism plays an important role, as some of the QUIC endpoints use RR schedulers while others employ sequential multiplexing.

### 3.3 HOL Blocking Results and Discussion

Tracking the `data_moved` events for QUIC and TCP produces two output lists, one for each protocol, their values indicating the amount of bytes blocked at each HOL blocking instance. Larger values indicate more data was held back and thus more HOL blocking was observed. To get a feel for this data, let's first consider the highest blocked byte amounts seen across all the traces. For TCP for example, the highest value seen was 3740019 bytes (in a *mvfst* trace), constituting over a third of the total amount of transmitted bytes on the connection (ten times 1MB). Conversely, the highest value encountered for QUIC was 969151 bytes (unsurprisingly in a sequentially scheduled *picoquic* trace), a factor of three smaller than TCP.

The full set of results is shown in Figure 3. Note that, as we use traces from experiments run over real networks, not every trace contains the same occurrences of packet loss or HOL blocking. As we are mainly interested in the worst-case differences that can occur between the protocols, we take the maximum blocked byte amount for each protocol. To be able to more easily compare TCP and QUIC measurements, we calculate the ratio between these two maximum values per trace:

$$ratio = \frac{max\_blocked\_bytes(QUIC)}{max\_blocked\_bytes(TCP)}$$

As such, ratios close to 1 indicate that the maximum values were very similar and QUIC suffered from HOL blocking more similarly to TCP. Conversely, ratios closer to 0 indicate that QUIC's maximum HOL blocked byte amount was (much) lower than TCP's, indicating it more frequently passes data to the application layer than the legacy transport protocol. Each trace's ratio is plotted as a point in Figure 3. As traces without packet loss would give a ratio of exactly 1 (as both maxima would be the packet size), these are left out of the results, explaining the low amount of results for *google* and *ats*. Interpreting Figure 3, we can see that for most endpoints and most traces, QUIC indeed achieves a much lower maximum amount of HOL blocked bytes than TCP, as most ratios are between 0.1 and 0.3. As predicted, the ratios shift upwards for the *picoquic* endpoint, as its sequential scheduler induces additional intra-stream HOL blocking compared to the other endpoints' RR schedulers. However, from these results alone it is difficult to assess the practical impact of QUIC's HOL blocking behaviour on application metrics such as page load times.

## 4 HTTP/2 Prioritization

Now that we understand how QUIC can approach general purpose multiplexing across different streams, it is time to look at how this behaviour can be influenced using semantics from the HTTP layer, which is necessary to achieve optimal page loading performance. While it may seem this is mainly important for H3, the concept of a prioritization system to dictate web page resource scheduling behaviour actually originates with the H2 protocol. This is because H2 includes its own stream abstraction and framing layer, also utilizing stream IDs in DATA frames for multiplexing [11]. This is incidentally one of the main reasons it is difficult to use H2 on

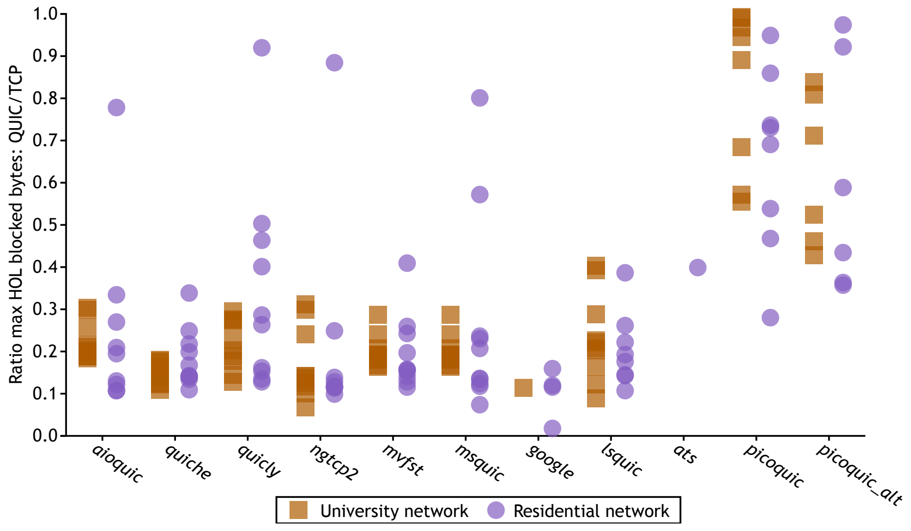

Fig. 3: Maximum HOL blocking ratios between TCP and QUIC. Each point is a single test run downloading ten 1MB files. Higher values mean more HOL blocking in QUIC.

QUIC directly, as this would lead to two separate and competing stream concepts, which can introduce much implementation complexity and inefficiencies. The choice was made instead to define a new mapping of H2 onto QUIC, which is now being called H3. In essence, the main change is that in H3, all of H2's stream-specific amenities have been removed in favor of utilizing QUIC's streams directly. This does mean that H3 still needs a prioritization system to steer QUIC and as such, it is interesting to first understand the details of H2's approach.

## 4.1 Background: The HTTP/2 Dependency Tree

In our discussion of the QUIC implementations' multiplexing behaviours in Section 2, it was clear they treated each resource and each stream in the same way. No individual requested file was considered more important than others; only retransmitted data was given a higher priority in some implementations. While this makes sense in the general purpose QUIC case, it does not when looking at HTTP semantics and the use case of loading a web page with multiple resources. Not all these resources are equally important and most have very distinct characteristics during the web page loading process. For example, HTML and image files can conceptually be parsed, processed and rendered incrementally. This is different from JS and CSS files, which can be parsed as data comes in (at least in some browsers) but have to be fully downloaded to be actually executed and applied. Another issue is that the browser does not know about all the needed resources up-font, as they are discovered incrementally during the page load and resources can import other files dynamically. Finally, users typically only see a small part of the web page at a time due to screen sizes, which makes resources that are currently in the viewport extra important. These and other web page loading concepts make that web pages can have very complex resource interdependencies. Individual resource importance depends on its type, precise function, (potentially) location within the HTML and how many children it will end up including. Relative resource priorities can also change over time, as new resources of a higher importance are discovered.

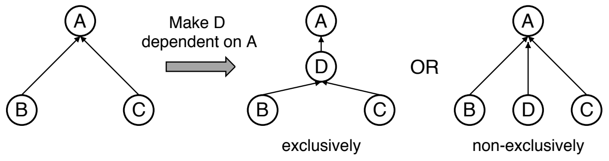

Fig. 4: HTTP/2 dependencies: exclusivity (source: [6])

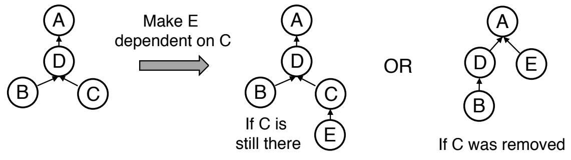

Fig. 5: HTTP/2 behaviour when referenced parent does not exist.
E is added as a sibling of D on the root, (unintentionally) sharing its bandwidth (source: [6])

To be able to manage these volatile requirements, HTTP/2 includes an advanced prioritization system in the form of a 'dependency tree', in which each individual resource stream is represented as a node. Available bandwidth is then distributed across these nodes by means of two simple rules: parents are transferred in full before their children, and sibling nodes share bandwidth among each other based on assigned weights. For example, given a sibling A with weight 128 (out of a maximum of 256) and a sibling B with weight 64, A will receive 2/3 of the available bandwidth, leaving 1/3 for B. In an optimal implementation, this would result in the following scheduled frame sequence: AABAABAAB.... As HTTP/2 assumes that, between the two endpoints, the browsers have the best view of relative resource importance, it has the user agents build and maintain this dependency tree over time, which is then synchronized back to the server by means of PRIORITY frames. As such, the structure of the tree evolves over time on both endpoints, as new resources are discovered and added, and fully transferred nodes are pruned.

H2 provides two main ways to add nodes as children to a parent in the tree: exclusively and non-exclusively. As can be seen from Figure 4, non-exclusive addition is the 'normal', less invasive way of adding nodes to the tree. Exclusive addition however, changes all of its potential siblings beneath its parent to instead become children of the newly added node itself. This allows aggressive (re-)prioritization, by displacing (large) groups of nodes in a single operation. One special case arises when the intended parent is no longer available at the server (i.e., because it was fully transmitted and was pruned from the tree). In this case, the child is added to the root node of the parent instead, leading to a possible desynchronization of the tree between client and server and a possible mis-prioritization of the new child, as it is now a sibling of other direct children of the root. An example of this can be seen in Figure 5.

While this setup in general seems easy enough to comprehend, it can be difficult to implement correctly, especially if the tree updates frequently. Additionally, the possibility of desynchronization between the two endpoints can lead to unintended behaviours. Furthermore, the tree needs to be evaluated on the server for each packet that is to be sent, to determine which stream should received bandwidth, which can be computationally expensive for large trees. Given this complexity and these edge cases, one might wonder why it was decided that the browser should determine the resource priorities instead of the server. Could we not make a similar argument that the server (usually) already has all the resources for the web page and thus has a good overview from the start? While letting the server dictate priorities is certainly possible, it is quite complex in practice to know the resource priorities at the server at the start of the page load [7]. Still, H2 provides an option for this approach as well: servers are free to ignore the clients' PRIORITY messages and to concoct a more optimized bandwidth distribution from server-side info.

### 4.2 Prioritization: Theory vs Practice

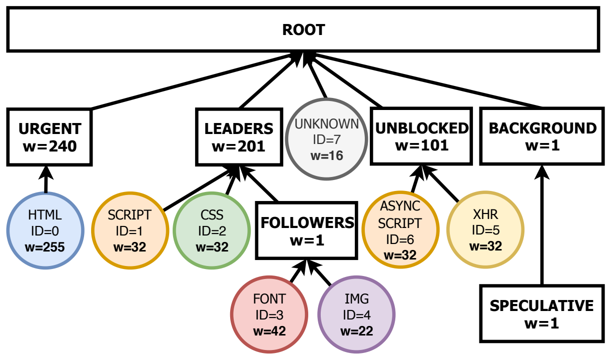

Fig. 6: Firefox's HTTP/2 dependency tree (source: [6])

Given the high flexibility provided by H2's prioritization system and the choice between client and server-side prioritization, it is interesting to see how these amenities are actually being used in practice by real-world implementations. [19] looked at how modern browsers utilize H2's prioritization system in practice. They found that out of 10 investigated browsers, only Mozilla's Firefox constructs a non-trivial dependency tree (see Figure 6). It uses "empty" nodes that are not tied to a real resource as placeholders to group other nodes and assigns heterogeneous weights. Other browsers like Chrome and Safari instead opt for much simpler schemes, the former creating a purely sequential model where all resources are added to a parent exclusively, the latter a RR variant where all resources are added non-exclusively to the root using different weights. The original Edge browser neglected to specify any priorities at all, relying on H2's default behaviour of adding all the resources to the root with identical weights, leading to pure RR.

Next, [2] looked at how various H2 servers actually adhere to client-side PRIORITY directives in practice. They find that out of 35 tested implementations, only 9 actually properly support (re-)prioritization. They posit this is due to faulty or inefficient implementations, servers ignoring client directives but failing to provide better server-side scheduling, and various forms of 'bufferbloat'. This last problem occurs when deployments use too large buffers: the risk exists that these buffers will be filled with low-priority data before the high-priority requests arrive [8]. This is similar to the problem encountered with *mvfst* in Section 2.3.

Next to this H2 specific work, there are also contributions on resource prioritization in general. WProf [17] looks at resource dependencies and their impact on total page load performance. Polaris [7], Shandian [18] and Vroom [14] collect very detailed loading information and construct complex resource transmission and computation scheduling schemes, claiming 34% - 50% faster page load times at the median. However, none of their implementations utilize H2's server-side prioritization, instead using JavaScript-based schedulers or H2 Server Push. At this time Cloudflare is the only commercial party experimenting with advanced server-side H2 prioritization at scale, for which they employ the *bucket* scheme from [9]. They claim improvements of up to 50% for the original Edge browser.

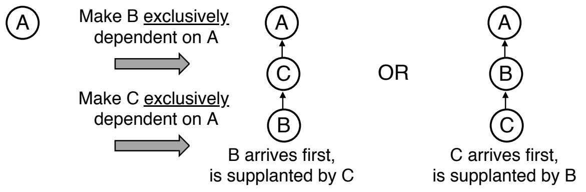
A
Make B exclusively dependent on A
Make C exclusively dependent on A
A
C
B
B arrives first, is supplanted by C
OR
A
B
C
C arrives first, is supplanted by B

Fig. 7: Non-deterministic ordering over QUIC:
intended structure of the dependency tree can become inverted (source: [6])

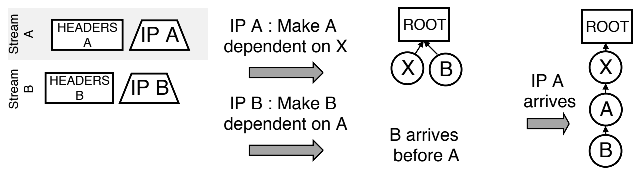

Fig. 8: Non-deterministic ordering over QUIC: B can
end up 'stealing' bandwidth from X for a time. (IP: Initial PRIORITY message) (source: [6])

Overall, we can conclude that the flexibility of H2's prioritization system is barely used in practice, while its complexity leads to several bugs in real world deployments. Additionally, advanced server-side prioritization remains relatively unproven in practice and many servers that ignore the client's PRIORITY messages do not employ optimal custom scheduling logic.

## 4.3 Adaptation for HTTP/3

Given the complexities inherent in H2's prioritization system and its abysmal adoption in practice, there were already some who questioned whether the approach should be reworked for H3. This was then further compounded by the fact that directly using H2's setup to run in H3 over QUIC would be impossible due to QUIC's non-deterministic ordering, if PRIORITY messages are sent on different streams. This can be simply understood by looking at a few examples. Firstly, consider the two opposite outcomes if two H2 PRIORITY messages, each carrying an exclusive parenting operation, would become re-ordered over QUIC, see Figure 7. Secondly, priority message re-ordering can lead to nodes being added to the default root node, if their intended parent had not yet been added to the tree, see Figure 8. This is similar to the problem from Figure 5. Several more similar edge-cases can occur within this setup.

Over time, the H3 designers debated multiple possible approaches and changes to H2's system to solve or at least alleviate these issues on H3. At one point, the proposed solution was to use a separate control stream to send priority updates. This would prevent them from becoming re-ordered, as messages within a single stream are of course delivered in-order. In tandem, a separate default parent node was used (termed the 'orphan' node) to prevent new nodes from being added to the root and stealing bandwidth from unintended siblings. This setup can be seen in Figure 9. While this setup indeed made the worst edge cases manageable, it was still not ideal. For example, there was a form of HOL blocking where resources would be mis-prioritized for a full network round trip if there was loss on the control stream carrying the PRIORITY messages. A much more in-depth discussion of these issues and contemplated solutions can be found in our previous work [6]. Due to this added complexity to an already complex system, several people proposed alternative solutions to prioritization for H3. These are discussed and evaluated in Section 5.

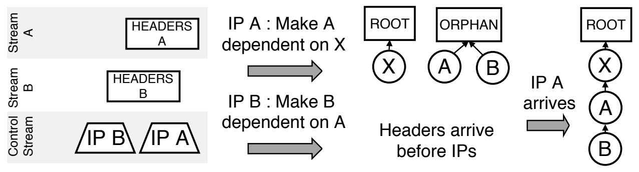

Fig. 9: Proposed H3 prioritization setup: control
stream and separate default parent (orphan). (IP: Initial PRIORITY message) (source: [6])

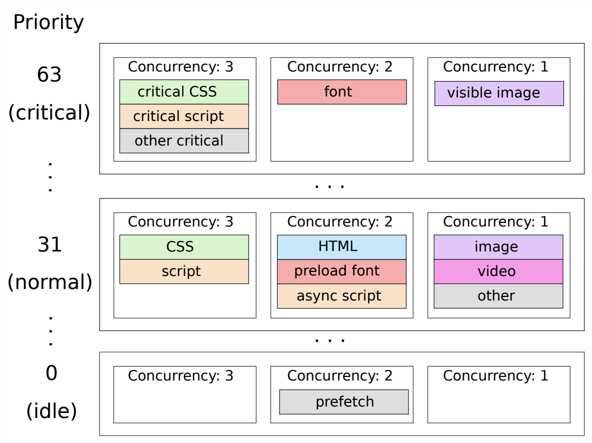

Fig. 10: Proposal for HTTP/3 prioritization based on priority buckets, from [10]

## 5 HTTP/3 Prioritization

### 5.1 Alternative Proposals

As many felt that the proposed integration of H2's dependency tree system in H3 was too complex, several alternative proposals were launched. Their first goal was trying to simplify the setup to make it easier to implement and reason about. A secondary goal was that it should be possible to back port the new approaches to H2 as well, to be able to fix the problems seen in current H2 deployments. A final goal was to make it easier to combine server-side information with client-side PRIORITY messages. As mentioned in Section 4.1, it is possible that a server has better information about a resource's relative importance than the client. However, in practice it is difficult for the server to integrate that knowledge into H2's dependency tree. As clients are free to build their tree in very different ways [19], it is almost impossible for a server implementation to automatically derive the semantics employed by various clients. This makes it difficult to determine the correct place in the tree for the (manually) prioritized resource. In practice for H2, servers either need to follow the client's setup, or ignore it completely and define a full new approach for all resources at the same time. As such, a new setup for H3 should ideally make it easier to combine client and server-side directives.

The first proposal, termed *bucket* by us, is one by Patrick Meenan from Cloudflare [10]. He proposes to drop the dependency tree setup and replace it with a simpler scheme of ‘priority buckets’, see Figure 10. Buckets with a higher number are processed in full before buckets with a lower number. Within the buckets, there are three concurrency levels. Level three, called

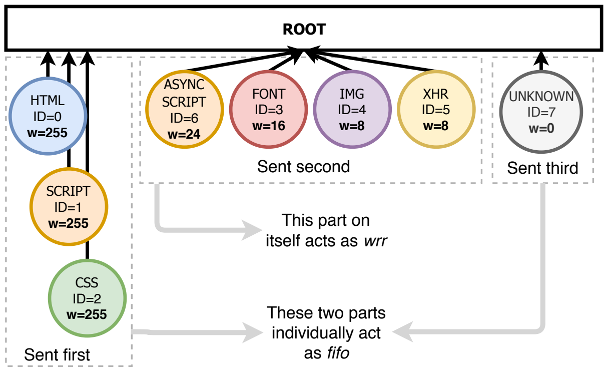

Fig. 11: Tree for our HTTP/3 zero weighting proposal (source: [6])

"Exclusive Sequential" preempts the other two and sends its contents sequentially by stream ID (streams that are opened earlier are sent first). Levels two ("Shared Sequential") and one ("Shared") are each given 50% of available bandwidth if level three is empty. Within level two, streams are again handled sequentially by lowest resource ID, while within level one, they follow a fair Round-Robin scheduler. As can be seen in Figure 10, this allows a nice and fine-grained mapping to typical web page assets loading needs. This scheme was deployed for H2 as well on Cloudflare's edge servers and they claim impressive speedups [9]. Overall, this scheme is also easier to implement than the dependency tree: all that is needed is a single byte per resource stream to carry the priority and concurrency numbers. Resources can easily be moved around by updating these numbers, though it is not possible to re-prioritize many resources at once. This scheme also makes it easier to incorporate server-side directives for particular resources, as there is a clear ordering of importance through the priority buckets.

Secondly, our own proposal[9] called '*zeroweight*' has an aim to stay close to the idea of a dependency tree, but to significantly reduce ways in which new nodes can be added. The main change is that nodes can now have a weight between 0 and 255 (where before it was in the range 1-256). Nodes with weight 0 and 255 exhibit special behaviour, akin to Meenan's sequential concurrency levels: siblings with weight 255 are processed first, in full and sequentially in the lowest stream ID order. Then, all siblings with weight between 254 and 1 are processed in a weighted Round-Robin fashion (assigned bandwidth relative to their weights, see Section 4.1). Finally, if all other siblings are processed, do zero-weighted nodes get bandwidth, again sequentially in the lowest stream ID order. The resulting tree can be viewed in Figure 11. This proposal requires just a few semantic changes to the H2 system, and is thus easy to integrate in existing implementations, while being much easier to implement for H3 than the more flexible dependency tree. This setup also also makes it easy to integrate server-side directives, as changing per-resource weights can now have more impact and it is straightforward to promote or demote a resource to a higher 'tier' (change weight to 255 or 0).

Thirdly, we shortly discuss several other proposals. For example, the 'strict priorities' approach from Ian Swett at Google[10] attempted to integrate the semantics of Patrick Meenan's *bucket* proposal with the dependency tree setup to achieve a 'best of both worlds' outcome. As it should perform similarly to *bucket*, we did not evaluate this setup ourselves. Another proposal was to return to the prioritization scheme of the SPDY protocol [15]. SPDY was the predecessor of H2

[9] github.com/quicwg/base-drafts/pull/2723
[10] github.com/quicwg/base-drafts/pull/2700

Table 1: Prioritization schemes. The top seven are from actual browser H2 implementations and [19]. The bottom four are new proposals for H3 (source: [6])

| **Name** | **Description** |
|---|---|
| *rr* (Edge) | Fully fair Round-Robin. Each resource gets equal bandwidth. |
| *wrr* (Safari) | Weighted Round-Robin. Resources are interleaved, but non-equally, based on weights. |
| *fifo* | First-In, First-Out. Fully sequential, lower stream IDs are sent in full first. |
| *dfifo* (Chrome) | Dynamic FIFO. Sequential, but higher stream IDs of higher priority can interrupt lower stream IDs. |
| *firefox* | Complex tree-based setup with multiple weighted placeholders and *wrr* for placeholder children. See Figure 6. |
| *p+* | Parallel+. Combination of *dfifo* for high-priority with separate *wrr* for medium and low-priority resources [19]. |
| *s+* | Serial+. Combination of *dfifo* for high and medium-priority with *firefox* for low-priority resources [19]. |
| *spdyrr* | Five strict priority sequential buckets, each performing *wrr* on their children. The Round-Robin counterpart of *dfifo*. |
| *bucket* | Patrick Meenan's proposal, Figure 10. |
| *bucket HTML* | Our variation on Patrick Meenan's proposal, with HTML having a higher priority (bucket 63 instead of 31 in Figure 10). |
| *zeroweight* | Our proposal, Figure 11. |

and had just "eight levels of strict priorities". The SPDY specification did not provide details on how resources should be allotted bandwidth, only that resources of higher priority levels should be sent first. In our evaluation in Section 5.3 we discuss a Round-Robin version of this setup, termed *spdyrr*, which processes the priority levels sequentially, but applies a fair RR scheduler between resources within a single level. More details on these and further proposals can be found in [16].

While at first glance these proposals seem to achieve the set goals, it is difficult to assess if they will indeed be able to provide similar or better web page loading performance when compared to the existing H2 setups seen in the different browser and server implementations. Without proof that these new setups would indeed be able to pull their own weight despite shedding considerable complexity, the H3 designers were reticent to drop the dependency tree. To aid their decision making, we implement and evaluate the different proposals.

## 5.2 Experimental Setup

In order to prove that the simpler H3 approaches can perform similarly or better than the existing H2 dependency tree configurations, we first need to determine a baseline for the performance of these different H2 schemes. While Wijnants et al. provide an extensive evaluation of these schemes [19], it is unclear if their findings also hold on H3 over QUIC. As such, next to the new H3 setups, we also implement and evaluate the main existing H2 schemes, resulting in a total of 11 different evaluated prioritization setups. Their main approaches are described in Table 1 and Figure 12 shows to what kind of data scheduling they lead in practice for an example page load. For example, as expected the fair Round-Robin *rr* clearly has a very spread out way of scheduling data for the various streams, as was also evident from similar approaches in our QUIC experiments. The *firefox, p+, s+ and spdyrr* schemes are quite similar, but include subtle differences. Looking at the results for *bucket* we see that the HTML resource (and the font that is directly dependent on it) are delayed considerably, which seems non-ideal. As such, we propose our own

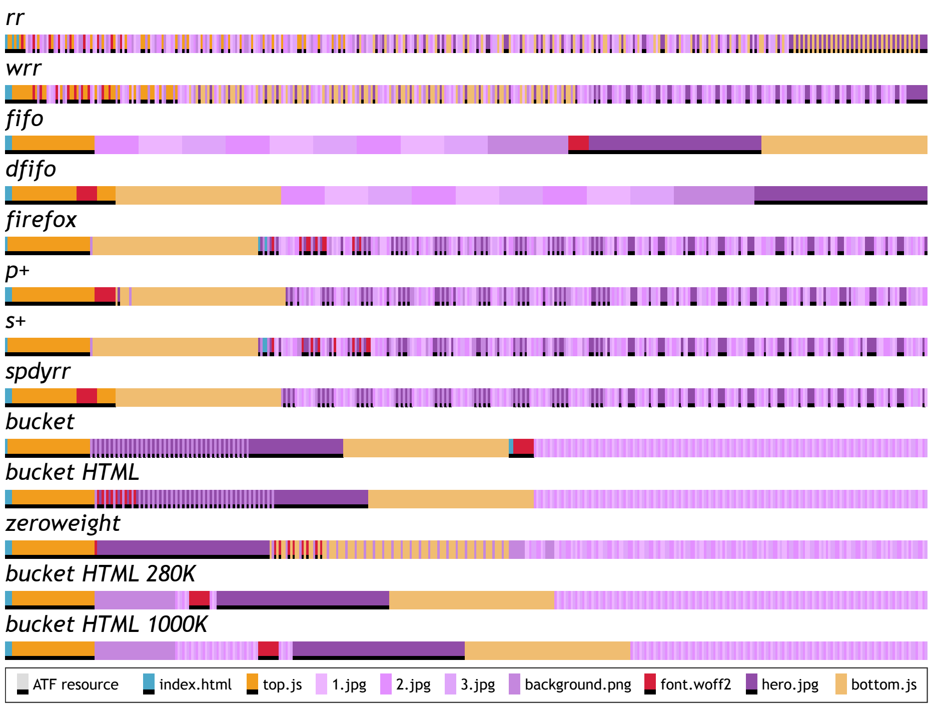

Fig. 12: Scheduling behaviour of various prioritization schemes for a single, synthetic test page from [2]. Each individual colored rectangle represents a single QUIC packet of 1400 bytes. Packets arrive at the client from left to right. The bottom two lines show results with non-zero send buffers. Resources in the legend are listed in request-order from left to right (source: [6])

variation, *bucket HTML*, which gives the HTML resource a higher priority. For this test page it dramatically shortens the HTML and font file's Time-To-Completion (TTC). Figures 6 and 11 show dependency tree layouts for two of the schemes; the rest can mostly be found in [19] and [16].

To make comparisons with earlier results easier, we test the 11 prioritization schemes on the test corpus of [19]. This corpus consists of 40 real web pages from the Alexa top 1000 and Moz top 500 lists. The corpus represents a good mix of simple and more complex pages (10-214 resources), as well as small and larger byte sizes (29KB-7400KB). We also add two synthetic test pages: one of our own design that tests all types of heuristics modern browses apply, and the one used by [2] (Section 4.2, Figure 12). These two pages can be seen as "stress-tests" and are designed to highlight prioritization issues and behaviour. The full corpus is downloaded to disk and all files are served from a single H3+QUIC server.

For this QUIC server, we choose the open source TypeScript and NodeJS-based Quicker implementation [13] because the high level language makes it easy to implement the prioritization schemes. We have exhaustively tested the implementation to make sure any inefficiencies stemming from the underlying JavaScript engine did not lead to performance issues. As this part of our evaluation was run for of our previous work [6], it was performed several months before our tests of the various QUIC implementations in Section 2.2. At that time, the Quicker implementation was one of the few with a fully functional H3 implementation which allowed us to experiment with prioritization. During the intervening months, Quicker however became outdated with respects to the other implementations, which is the reason it was not evaluated in our tests for the QUIC implementations discussed earlier.

On the client side, there is currently sadly no browser available that (fully) supports H3. As such, we use the Quicker command line client instead. However, we do closely emulate the browser's expected behaviour by using the open source WProfX tool[11], an easy to use implementation of the concepts from the original WProf paper [17]. We host the test corpus on a local H2O optimized webserver and load the pages via the Google Chrome-integrated WProfX software. From this load, the tool can extract detailed resource inter-dependencies (e.g., was an image referenced in the HTML directly or from inside a CSS file) and request timing information. Our H3 Quicker client then performs a "smart play-back" of the WProfX recording, taking into account resource dependencies (e.g., if the current prioritization scheme causes a CSS file to be delayed, the images or fonts it references will also be delayed accordingly). The tool also indicates which resources are on the "critical path" and are thus most important to a fast page load.

At the time of our evaluation, none of the open source QUIC stacks (including Quicker) integrated a performant congestion control implementation that had been shown to perform on par with best in class TCP implementations. As we want to focus on the raw performance of the prioritization schemes and the order in which data is put on the wire, we do not want to run the risk of inefficient congestion controllers skewing our results. We instead manually tune the Quicker server to send out a single packet of 1400 bytes containing response data of exactly one resource stream every 10ms (i.e., simulating a steadily paced congestion controller). To see if we can replicate the results of [8], we also implement the option to use small and larger application-level send buffers, to assess the impact of "bufferbloat". Given these factors, our results represent an "ideal" upper bound of how well prioritization could perform in the absence of network congestion and retransmits. This approach also leads to exceptionally stable experimental conditions, with re-runs of individual experiments leading to near-identical results.

Due to our stable experimental setup we can not simply use, for example, the total web page download time as our metric, as these values are all identical per tested page across the different schemes. This can easily be seen by understanding that each scheme still needs to send the exact same amount of data; it just does so in a different order. Instead, we will mainly look at so-called "Above The Fold" (ATF) resources. These are the resources that are either on the browser's critical render path (meaning they would delay the load and usage of other resources) or that contribute substantially to what the user sees first (e.g., large hero images). We combine WProfX's critical path calculations with a few manual additions to arrive at an appropriate ATF resource set for each test page. This ATF set typically contains the HTML, important JS and CSS, all fonts and prominent 'hero images'. Non-hero (e.g., background) images that are rendered above the fold are consciously not included in this set (e.g., see "background.png" in Figure 12), as they should have less of an impact on user experience. Furthermore, to highlight the power that comes from combining client and server-side directives, our implementations of both *bucket* and *zeroweight* use small parts of these ATF resource lists to simulate explicit manual web developer prioritization interventions. Concretely, the hero images are given a higher server-side priority than what they would normally receive from the client. For example, Figure 10 mentions a 'visible image' for the *bucket* scheme, while in practice, browsers have no way of definitively knowing which images will eventually be visible or not. Since the other discussed schemes do not utilize this additional metadata, this will in part explain the seemingly best-in-class performance of *bucket* and *zeroweight* in our results.

[11] wprofx.cs.stonybrook.edu

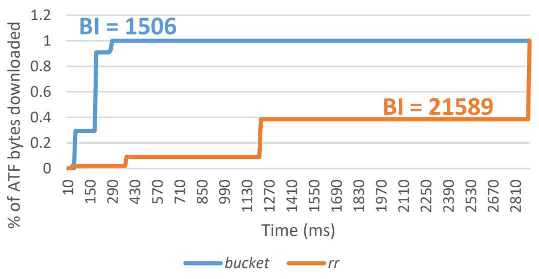

Fig. 13: ByteIndex (BI) for *bucket* and *rr* schemes. *Bucket* is clearly faster for ATF resources. Looking at these schemes in Figure 12, it is immediately clear why (source: [6])

However, to report these results, we also cannot directly use, for example, the mean TTC for these ATF resources as our metric. For example, receiving most of the ATF files very early and then receiving just a single one late is generally considered better for user experience than receiving all together at an intermediate point, though both situations would give a similar mean TTC. To get a better idea of the progress over time, we use the ByteIndex (BI) web performance metric [1]. This metric estimates (visual) loading progress over time by looking at the TTCs of (visually impactful, e.g., ATF) resources. At a fixed time interval of 100ms we look at which of the resources under consideration have been **fully** downloaded. The BI is then defined as taking the integral of the area above the curve we get by plotting this download progress, see Figure 13. Consequently as with normal web page load times, lower BI values are better. Practically, we instrument Quicker to log the full H3 page loads in the *qlog* format[12]. We then write custom tools to extract the needed BI values from these logs, as well as new visualizations to display and verify our results (Figures 12 and 14).

## 5.3 Prioritization results

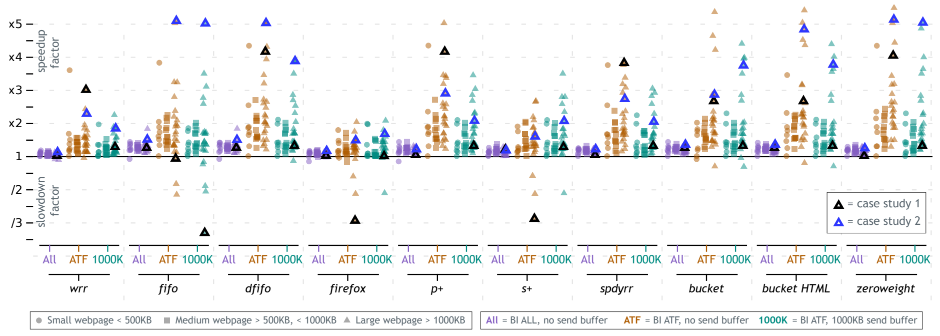

Fig. 14: ByteIndex (BI) speedup and slowdown ratios for 10 prioritization schemes compared to the baseline *rr* scheme. Each datapoint represents a single web page, split out by total page byte size. Higher y values are better (source: [6])

Our main results are presented in Figure 14 and Table 2. Like [19], when processing the results we quickly saw that the *rr* scheme is by far the worst performing of all tested setups, with almost

Table 2: Mean speedup ratios compared to *rr*
per other prioritization scheme from Figure 14. Higher mean values are better (source: [6])

| Name | Mean All | Mean ATF | Mean 1000K |
|---:|:---:|:---:|:---:|
| *wrr* | **1.05** | 1.49 | **1.28** |
| *fifo* | **1.27** | 1.93 | 1.57 |
| *dfifo* | **1.27** | 2.30 | 1.72 |
| *firefox* | 1.07 | **1.22** | 1.25 |
| *p+* | 1.17 | 2.20 | 1.64 |
| *s+* | 1.14 | 1.45 | 1.56 |
| *spdyrr* | 1.14 | 1.96 | 1.57 |
| *bucket* | 1.20 | 2.13 | **1.82** |
| *bucket HTML* | 1.20 | **2.49** | **1.83** |
| *zeroweight* | 1.15 | **2.8** | **1.9** |

no data points performing worse. This is mainly because for many high-priority ATF resources (e.g., JS, CSS, fonts) it is imperative that they are downloaded **in full** as soon as possible. As can be seen at the top of Figure 12, RR bandwidth interleaving leads to resource downloads being completed very late. As such, a more sequential scheduler, which sends a single resource at a time, is a better approach for many types of critical resources, while a RR scheme is more apt for lower-priority resources that can be incrementally used (e.g., progressive images).

Consequently, we take *rr* as the baseline and present the other measurements in terms of a relative speedup to that baseline result. As such, a speedup of x2 for scheme Y means that, for a baseline *rr* BI of 1500, Y achieves a BI of 750. Symmetrically, a slowdown of /3 indicates that Y had a BI of 4500. We have tested the schemes with application-level send buffers of 14KB (about 10 packets and similar to the default minimum H2 frame size of 16KB [11]), 280KB and 1000KB, but found that these had relatively small effects until the buffer grows substantially large. As such, we focus on results for send buffers of 1000KB here. As seen from the *mvfst* example in Section 2.3, this is a realistic value.

A few things are immediately clear from Figure 14: a) Almost all data points are indeed faster than *rr*. b) With the exception of a few bad performers (i.e., *firefox, wrr, s+*), all schemes are able to provide impressive gains of x3.5 to x5+ speedup factors for individual web pages. c) Medium sized pages seem to profit less from prioritization overall, with smaller and larger pages showing higher relative advancements. d) Of the well-performing schemes, there is not a clear, single winner or a scheme that consistently improves heavily upon *rr* for -all- tested pages. e) The impact of the 1000KB send buffer is visible, but less impressively so than perhaps indicated by previous work [8], which quoted slowdowns of /2 compared to small/non-existent send buffers.

When looking at the mean ratios in Table 2, we see similar trends. We have highlighted some of the the highest and lowest values for each column. Taking into account all page assets, even though the speedups are all modest, it is clear that *fifo* is a far better default choice than *rr*. Looking at ATF resources only, it is remarkable how badly some schemes implemented by browsers perform (i.e., *firefox* and Safari's *wrr*), while Chrome's *dfifo* is almost optimal, after *bucket HTML* and *zeroweight*. Though all schemes suffer from larger send buffers, *bucket HTML* and *zeroweight* again come out on top. As mentioned before, this good performance of these latter two schemes can be partially attributed to giving hero images a higher server-side priority, highlighting that indeed, there might be merit in combining client and server-side directives.

While the reduced observed impact of larger send buffers might seem unexpected and contrary to the findings of [2], it has a simple explanation in two parts. Firstly, larger send buffers mainly impact the ability of the scheme to re-prioritize its scheduler in response to late discovered but important resources. In our data set however, we seem to have few web pages that contain such highly important late discoveries. Indeed, the test page showing the most remarkable slowdown from the larger send buffers was that introduced in [2] itself (dropping from x9 speedup without send buffer to x3 with 1000K). Secondly, as the size of the send buffer grows, the resulting behaviour more and more becomes that of *fifo*, as requested resources can be put into the buffer in their entirety immediately (similar to *mvfst*'s flow control behaviour in Section 2.3). This is clearly visible in Figure 12. As we have seen, *fifo* performs well overall, so even larger send buffers will also keep performing relatively well. It is our opinion that the results seen in [2] for faulty prioritizations in the wild might be less due to 'bufferbloat' and more due to misconfigured or badly implemented H2 servers, or to their choice of a highly tuned test page.

To dig a bit deeper into some of the outliers, we discuss two case studies. The first is outlined in black on Figure 14. This web page suffers a slowdown of about /3 for three separate schemes, yet sees major improvements of x4 in others. This specific page has relatively few resources with highly specific roles. Most importantly, it features a single, page-spanning hero image that is relatively small in byte size. Next, it includes several very large JS files which, even though included in the HTML `<head>`, are marked as "defer". This means they will only execute once the full page has finished downloading. As such, the hero image is marked as an ATF resource, but the JS files are not. As the image is discovered after the JS files, it is stuck behind them in *fifo*. For *firefox* (and similarly *s+*), the image is in the "FOLLOWERS" category (see Figure 6), while the JS files are in "UNBLOCKED". While the group of the image receives about twice the bandwidth as the JS (via the parent "LEADERS" placeholder), the image is competing with a critical CSS in the leaders, thus being delayed. For the speedups, the schemes either know there is a hero image (*bucket (HTML)* and *zeroweight*), allow the smaller hero image to make fast progress via a (semi) Round-Robin scheme or, in the case of *dfifo*, accurately assign low priority to the JS files.

The second case study is outlined in blue on Figure 14. This web page interestingly has a few instances where the 1000k send buffer outperforms the normal ATF case. This is because this page's HTML file is comparatively very large (167KB). As explained before, a large send buffer exhibits *fifo*-alike behaviour. Thus, for schemes where normally the large HTML would be competing with other resources (e.g., *bucket* and *firefox*), it now gets to fill the send buffers in its entirety, completing much faster. Where in the previous case study Round-Robin-alike schemes led to smaller resources completing faster, here the large HTML file is instead smeared out over a longer period of time due to interleaving with the other (ATF) resources.

## 6 Conclusion

### 6.1 HTTP/3's New Priority System

Based on our evaluation of the different prioritization schemes in 5.3, we can draw two general conclusions to help guide the choice for an appropriate H3 prioritization approach. Firstly, that it is perfectly possible to switch to a simplified prioritization framework while still fully supporting the web browsing use case and without losing performance. Schemes such as *bucket HTML* and *zeroweight* are easy to implement performantly without a dependency tree structure and seem to provide a good baseline performance for most web pages.

Secondly, that such a simpler scheme should nevertheless still allow enough flexibility. As our results and case studies have clearly shown, no single scheme performs well for all types of web pages. This is a conclusion we and related work keep repeating: it is almost impossible to come up with a perfect general purpose scheme. This is emphasized by the good results achieved by combining client-side priority indicators with server-side priority information in our *bucket HTML* and *zeroweight* schemes. This is also why efforts such as 'Priority Hints'[12] give developers options to manually indicate per-resource priorities. As such, any new chosen system should allow the integration of similar client and server-side overrides and behaviour tuning on a per-page basis.

These prioritization results and conclusions were first presented in our previous work [6] in June 2019, and were brought to the attention of the IETF QUIC and HTTP working groups. Based partly on our input and other insights, it was finally decided to remove the dependency tree setup from H3 completely and to instead develop a new system. As the alternative proposals tested in our evaluation were still deemed to be too complex, an even simpler proposal named "Extensible Prioritization Scheme for HTTP" [5] was adopted. This new proposal is simple in that it defines resource priorities based on just two parameters, termed 'Urgency' and 'Incremental'. The Urgency parameter is defined as an integer of value between 0 and 7. Intuitively, these values can be seen as individual priority buckets or levels, similar to the *spdyrr* scheme. These buckets are intended to be given bandwidth from low to high. The Incremental boolean parameter then defines whether resources sharing the same Urgency level should best be sent using a sequential (Incremental = 0) or an RR (Incremental = 1) scheduler. As such, unlike in the dependency tree setup, a resource's priority is no longer directly dependent on its relationship to other resources (parent or child), but is rather defined in a standalone, declarative fashion.

While simple in nature, this setup nevertheless provides all the benefits we had envisioned. Firstly, it can emulate (simplified versions of) both the *zeroweight* and *Bucket HTML* schemes. Secondly, it makes it easy to incorporate server-side overrides, as this only requires changing the Urgency and/or Incremental parameter values. Similarly, this makes it easy to re-prioritize resources. Thirdly, it can easily be emulated by a dependency tree in existing H2 deployments. Fourthly, it is easy to extend with more semantic parameters later (e.g., to indicate the resource is RenderBlocking or OnScreen) to provide more fine-grained scheduling options.

At the time of writing, implementation of this new scheme is expected to start soon in several HTTP/3 implementations and thus an evaluation of this setup is left for future work.

## 6.2 Transport Layer Multiplexing

Evaluating both QUIC transport layer (Section 2.3) and HTTP/3 application layer (Section 5.3) behaviours allows us to draw conclusions about their possible interplays.

The problems that might arise are most evident in our discussions of Round-Robin schedulers. As we have shown for H3, these can lead to worst case web page loading performance, as they can delay the full download of key resources compared to a sequential scheduler. Consequently, it is somewhat counter-intuitive to find that RR was the default for H2 and that a majority of QUIC implementations in fact implements RR as their default scheduler. The latter could be explained by the fact that QUIC can be used as a general purpose transport protocol, and is probably implemented that way in most stacks. As such, it should indeed not just be tuned for HTTP/3, as other use cases might in fact prefer Round-Robin schedulers. Still, this again highlights the need for clear and flexible QUIC-level prioritization APIs to allow H3 implementations running on top to specify more sequential behaviours as needed. As discussed in Section 2.3, these APIs are currently not well defined, which might lead to problems down the line.

[12] github.com/WICG/priority-hints

Somewhat contradictory to the previous paragraph is that, while we have found RR to perform badly for loading web pages on lossless networks, we have also seen it is the optimal approach for reducing transport layer HOL blocking. This is in opposition to more sequential schemes, which seem better for web page loading, but are more at risk of becoming HOL blocked. However, it is unclear how much of an impact HOL blocking actually has on web page loading, as we only explored its behaviour on the transport layer. Future work is needed to explore the impact of the complex combination of congestion control, packet loss and application layer multiplexing behaviour.

## References

1. Bocchi, E., De Cicco, L., Rossi, D.: Measuring the quality of experience of web users. In: Proceedings of the 2016 Workshop on QoE-based Analysis of Data Communication Networks. pp. 37–42. Internet-QoE '16, ACM (2016)
2. Davies, A., Meenan, P.: HTTP/2 priorities test page. Online, https://github.com/andydavies/http2-prioritization-issues (December 2018)
3. De Coninck, Q., Michel, F., Piraux, M., Rochet, F., Given-Wilson, T., Legay, A., Pereira, O., Bonaventure, O.: Pluginizing quic. In: Proceedings of the ACM Special Interest Group on Data Communication. pp. 59–74. ACM (2019)
4. Iyengar, J., Thomson, M.: Quic: A udp-based multiplexed and secure transport. Internet-Draft 24, IETF Secretariat (November 2019), http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-24
5. Kazuho Oku, Lucas Pardue: Extensible Prioritization Scheme for HTTP. Online, https://github.com/httpwg/http-extensions/blob/master/draft-ietf-httpbis-priority.md (December 2019)
6. Marx, R., De Decker, T., Quax, P., Lamotte, W.: Of the utmost importance: Resource prioritization in http/3 over quic. In: Proc. of the 15th International Conference on Web Information Systems and Technologies: WEBIST,. pp. 130–143. INSTICC, SciTePress (2019)
7. Netravali, R., Goyal, A., Mickens, J., Balakrishnan, H.: Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In: Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation. pp. 123–136. NSDI'16 (March 2016)
8. Patrick Meenan: Optimizing HTTP/2 prioritization with BBR and tcp_notsent_lowat. Online, https://blog.cloudflare.com/http-2-prioritization-with-nginx (October 2018)
9. Patrick Meenan: HTTP/2 priorities test page. Online, https://blog.cloudflare.com/better-http-2-prioritization-for-a-faster-web (March 2019)
10. Patrick Meenan: HTTP/3 prioritization proposal. Online, https://github.com/pmeenan/http3-prioritization-proposal (February 2019)
11. RFC7540: HTTP/2. Online, https://tools.ietf.org/html/rfc7540 (May 2015)
12. Robin Marx: qlog logging format. Online, https://github.com/quiclog/internet-drafts (October 2019)
13. Robin Marx, Tom De Decker: Quicker: TypeScript QUIC and HTTP/3 implementation. Online, https://github.com/rmarx/quicker (June 2019)
14. Ruamviboonsuk, V., Netravali, R., Uluyol, M., Madhyastha, H.V.: Vroom: Accelerating the mobile web with server-aided dependency resolution. In: Proc. of the ACM SIG on Data Communication. pp. 390–403. ACM (2017)
15. SPDY: SPDY Protocol. Online, https://www.chromium.org/spdy/spdy-protocol (2014)
16. Swett, I., Marx, R.: IETF QUIC interim: HTTP/3 priorities status update. Online, https://github.com/quicwg/wg-materials/blob/master/interim-19-05/priorities.pdf (May 2019)
17. Wang, X.S., Balasubramanian, A., Krishnamurthy, A., Wetherall, D.: Demystifying Page Load Performance with WProf. In: Proceedings of the USENIX Conference on Networked Systems Design and Implementation. pp. 473–486. NSDI'13 (April 2013)
18. Wang, X.S., Krishnamurthy, A., Wetherall, D.: Speeding Up Web Page Loads with Shandian. In: Proc. of the 13th USENIX Conference on Networked Systems Design and Implementation. pp. 109–122. NSDI'16 (March 2016)
19. Wijnants, M., Marx, R., Quax, P., Lamotte, W.: Http/2 prioritization and its impact on web performance. In: Proceedings of the 2018 World Wide Web Conference. pp. 1755–1764. WWW '18, ACM (2018). https://doi.org/10.1145/3178876.3186181, https://doi.org/10.1145/3178876.3186181